Parallelizing Sequential Algorithms for the Generalized Assignment Problem

(Extended Abstract)

Ivan Yanasak[*], Gautam Shah[*], Zonghao Gu[†], Chris DeCastro[‡], Kalyan Perumalla[*], Yinhua Wang[†]
Anand Sivasubramaniam[*], Aman Singla[*], Martin Savelsbergh[†], Umakishore Ramachandran[*]
H. Venkateswaran[*]and Sudhakar Yalamanchili[‡]

Technical Report GIT-CC-94/44
September 1994

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

[*]College of Computing
[†]School of Industrial and Systems Engineering
[‡]School of Electrical and Computer Engineering

1 Introduction

The Generalized Assignment Problem (GAP) asks for a maximum profit assignment of n tasks to m agents such that each task is assigned to precisely one agent subject to resource restrictions on the agents. Although interesting and useful in its own right, its main importance stems from the fact that it appears as a substructure in many models developed to solve real-world problems in areas such as vehicle routing, plant location, resource scheduling, and flexible manufacturing.

The GAP is easily shown to be NP-hard. Therefore, optimization algorithms have to rely on some form of enumeration, usually branch-and-bound. The various algorithms algorithms that have been developed for the GAP mainly differ in the way they obtain bounds. Although substantial progress has been made since the first algorithm for the GAP was developed by Ross and Soland [RS75], the problem sizes that can be handled today are still relatively small.

We consider two state-of-the-art sequential algorithms for the GAP. The first has been developed by Karabakal, Bean, and Lohmann [KBL92] and applies Lagrangian relaxation to obtain upper bounds. A steepest descent multiplier adjustment method is used to solve the Lagrangian dual. The algorithm extends earlier work of Fisher, Jaikumar and Van Wassenhove [FJvW86] and Guignard and Rosenwein [GR89]. The second has been developed by Savelsbergh [Sav93] and applies decomposition principles to obtain a set partitioning formulation and solves the LP relaxation of this formulation to obtain upper bounds. Although in theory the bounds used by both algorithms are identical, on a sequential computer, the latter algorithm is clearly superior.

In this paper, we investigate whether the inherent parallelism of the branch-and-bound paradigm can be exploited successfully to achieve significant performance increases in both algorithms. Performance increases may come from evaluating tasks in parallel as well as from earlier pruning of the search tree.

In Section 2 we outline our project goals and in Section 3 we elaborate on the sources of parallelism in the two algorithms that we consider. In Section 4 we present the implementation details and computational results. The computational results demonstrate that parallelization provides both substantial speedup over sequential run-times and allows larger problem instances to be solved. Finally, in Section 5 we present some observations relating to parallelization of branch-and-bound algorithms, from an operations research perspective as well as from a parallel computing perspective.

2 Project Overview

As mentioned earlier, the algorithms we consider apply the branch-and-bound paradigm to implicitly enumerate all feasible solutions. Branch-and-bound algorithms generate search trees in which each node corresponds to a subset of feasible solutions. A subproblem associated with a node is either solved directly, or its solution set is partitioned and for each subset a new node is added to the tree. The process is enhanced by computing a bound on the solution a node can produce. If this bound is worse than the best solution found so far, the node cannot produce a better solution and hence can be excluded from further examination. Parallel activity is achieved by simultaneously evaluating several nodes. We will often use the term *task* to represent the evaluation of a single node. Observe that in a branch-and-bound algorithm there is a minimum set of tasks that has to be completed in order to prove optimality - these tasks constitute the nodes of the *critical tree*.

As far as this basic structure of computation is concerned, the two methods that we consider differ from each other in two primary ways for any instance of nontrivial size: (1) the linear programming based algorithm explores fewer tasks than the Lagrangian relaxation based algorithm, but (2) the linear programming based algorithm requires more time to evaluate a task than the Lagrangian relaxation based algorithm. These

characteristics suggest that our linear programming based implementation will be more suited to "coarse-grained" hardware platform, whereas our Lagrangian relaxation based implementation will benefit from a more "medium-grained parallelism" hardware platform.

Our primary goals in attempting to parallelize the GAP were two-fold.

1. We wanted to be able to solve instances with sizes that are not computationally viable in the sequential environment.

2. We wanted to explore various approaches to parallelize the branch-and-bound algorithm and implement a scalable scheme which shows good speedups.

In addition, we wanted to investigate how easy (or hard) it is to parallelize the two sequential branch-and-bound algorithms and to study if their inherent differences make one of them more amenable to parallelization than the other.

To these ends, we implemented our own version of the linear programming based branch-and-bound algorithm for a cluster of 6 Sun SPARC-1 and SPARC-2 workstations, using the PVM [Sun90] message passing library and the CPLEX linear programming library [Inc93], and parallelized the Lagrangian relaxation based branch-and-bound algorithm of Karabakal [Kar92] for the 64-node KSR-2 [Res92] (a cache coherent cache only memory architecture shared memory machine). We tested both algorithms on a small set of random instances, generated according to distributions suggested in Martello and Toth [MT90]. We have only experimented with instances in class D, in which there is a correlation between objective values and knapsack weights, since this class contains the most difficult instances.

The remainder of this abstract presents the design issues and computational results for our PVM and KSR implementations.

3 Solution Approach

The algorithms we consider have two sources of parallelism. First, the computation of the upper bounds involves the solution of multiple independent knapsack problems (*intra-node parallelism*). Multiple processors can be used to solve the knapsack problems simultaneously to speedup the upper bound computation. Secondly, the branch-and-bound paradigm involves the solution of many independent subproblems (*inter-node parallelism*). Multiple processors can evaluate distinct subproblems simultaneously to speedup the search process. Note that the use of one source of parallelism does not preclude the use of the other and both can be used in conjunction, if one has the necessary computational resources.

We briefly discuss the trade-offs between exploiting just intra-node parallelism, exploiting just inter-node parallelism, and exploiting both intra- and inter-node parallelism that motivated our design choices. If we choose just intra-node parallelism, we have to consider the costs that result from state sharing, i.e., the computation to communication ratio should be high, so that the gain in computation time is not lost. Also there should be "enough" parallelism within each node for this method to be effective. The factors affecting inter-node parallelism are the cost of distributing the nodes to the various processors and the presence of a "sufficient" number of nodes at any given instant (the breadth of the search tree) so that processors are not idling. Inter-node parallelism could benefit from the faster pruning of the tree, because multiple paths are being explored in parallel.

Since we have a low to medium number of processors available for both implementations, we wanted to use the source of parallelism that would be most beneficial, given our resources. A profiling of the sequential code indicated that a sufficient number nodes were available at any given instant to successfully exploit inter-node parallelism. Furthermore, the time to evaluate a node is relatively small, so that implementation costs

for managing multiple processors to compute upper bounds might be a factor even if all other factors are beneficial for intra-node parallelism. Thus, as a first-cut, we chose to implement inter-node parallelism only. However, larger problem instances may benefit from exploiting both sources of parallelism, because for larger instances the per node time increases substantially. Therefore, parallelization of the multiple independent knapsack problems within each node is one of the future directions of our work.

A key detail in the implementation of inter-node parallelism is the distribution of tasks to the various processors. It is a common belief that for sequential branch-and-bound algorithms a best-bound search of the tree gives the best performance. Consequently, an efficient implementation of a branch-and-bound algorithm needs to maintain a priority queue of tasks. There are many choices for maintaining such a queue in a distributed setting but we observed that a simple centralized queue performed well and there were no bottlenecks caused by this queue access. This situation could change if there are a many more processors accessing the queue and hence greatly increasing contention. Since most of the total execution time ($> 85\%$) is spent in node evaluation, a sophisticated distributed queue implementation will not significantly change the overall costs. However, since a distributed queue could lead to lesser contention and hence better overall performance, it may be worth considering such an implementation.

We now discuss the specific details and computational results of each of our implementations.

4 Implementation

4.1 PVM Implementation

The LP-based sequential algorithm of Savelsbergh [Sav93] uses a column generation scheme to solve the linear program. In his implementation, all generated columns are kept in memory for the duration of the execution of the algorithm, and at each node only the relevant columns are used. In our implementation, none of the columns are permanently kept in memory, and at each node all relevant columns have to be regenerated. We decided upon this scheme for two reasons. First, maintaining a "global column pool" accessible to all PVM processes in the message-passing PVM environment would involve substantial memory and communication overheads, and would greatly increase the complexity of the parallel program. Secondly, we felt that maintaining local column pools would not be of obvious benefit, since the algorithm does not assume any relation between successive nodes evaluated by a given PVM process. (After our computational experiments, we now feel that local pools may be of some benefit, and implementing such a scheme is one of future directions of our work.) We rejected outright the possibility of sending the entire column pool with each node message, because it would require an extreme amount of memory and would greatly increase the communication costs. We discuss the consequences of our decisions in the next section.

For reasons of better potential scalability, our first PVM version utilizes a distributed node-queue scheme. After initialization by a process designated the "host", each task (one per workstation) takes a node message (if any) from its local "best-bound" queue (which maintains unevaluated nodes in order of nonincreasing bound values), evaluates the node (solves the LP using a column generation scheme), then, in case of an integral solution, broadcasts the new lower bound to all other PVM processes (to allow them to prune their node queues if possible), or, in the case of a fractional solution, places one of the two child nodes in its local queue and sends the other child node to another PVM process for placement in its queue. The recipient of the second child node is chosen at "random" (weighted so that processes on faster workstations are given preference over processes on slower workstations). This method is suggested by Karp and Zhang [KZ88] and provides relatively good load-balancing.

After running several 10x50 class D test instances, we noticed that in general the slower machines were terminating later than the faster machines, and that the idle times of the faster machines were much higher

Problem	Sequential	Distributed Send 1 Child		Distributed Send both		Centralized	
		Time	Speedup*	Time	Speedup*	Time	Speedup*
1	[a]	5737	-NA-	6098	-NA-	5275	-NA-
2	3574	1484	2.40	1156	3.09	876	4.07
3	3278	956	3.42	1099	2.98	916	3.57
4	9034	2198	4.11	2665	3.38	2456	3.67
5	11985	3237	3.70	3328	3.60	3114	3.84

* Speedup is with respect to the sequential time

[a] Could not complete due to insufficient memory

Table 1: Execution Times (in seconds) for 10x50 Class D Problems using 6 Processors

Problem	Keep Columns	Don't keep Columns	Slowdown Factor
1	1220	3334	2.73
2	269	1067	3.97
3	155	975	6.29
4	1876	2894	1.54
5	1385	3465	2.50

Table 2: Execution Times (in seconds) for the problems on the RS/6000 (sequential)

than those of the slower ones. To ensure that keeping one child node is not biasing our load balancing method, we also created a second distributed queue version that sends both child nodes to other PVM processes. This new "send both" scheme did not improve the execution times in general. We therefore created a third PVM version around a centralized queue process. This effectively prevents any load imbalance (at the cost of a small increase in processor idle time), as well as making termination detection more straightforward.

4.1.1 PVM Results

The resulting times for our PVM implementations, using 6 processors, are shown in Table 1.

As the data in the table shows, all of the parallel implementations provide significant speedup over the sequential times. Note, however, that no consistent difference in total time can be seen between the parallel implementations. In the "distributed versus centralized queue" this seems to indicate that time gained by ensuring that "fast machines finish last" is not all that substantial. The lack of a clear winner between the distributed queue methods indicates that keeping one child local is not currently a benefit or a disadvantage. Given that an average of 85-95% of the total execution times were spent in the LP and knapsack code, the addition of one extra send per node evaluation is expected to have minimal impact.

As discussed above, our implementation does not keep column data between node evaluations, but rather recomputes necessary columns for each node. The original sequential algorithm kept all columns around (marking those currently unusable) for the lifetime of the run. The two methods were compared (using the same problem set) on an IBM RS/6000 and the results are given in Table 2.

The results clearly demonstrate that not keeping the columns around between node evaluations incurs a substantial amount of overhead due to recomputation of old columns. Although for reasons that we mentioned earlier, keeping a global column pool is not feasible on our PVM platform, this data does suggest that an implementation on a shared memory machine (like the KSR-2 or a network of workstations that provides a distributed shared memory abstraction), which could use a shared column pool with much greater efficiency, would likely result in even greater speedups.

4.2 KSR Implementation

In the KSR implementation, aided by the support for shared memory, we only considered a centralized queue implementation in which each processor takes nodes from and updates a shared priority queue. In order to ensure that processors do not interfere with one another, we serialized access to the priority queue with mutual exclusion locks. The other major modification was to ensure termination of the algorithm.

4.2.1 KSR Results

Example runs of four 10x30 class D problems are shown in Table 3. As shown here spreading the work among

Number of Processors	Problem Number							
	1		2		3		4	
	Time	Queue*	Time	Queue*	Time	Queue*	Time	Queue*
1	550.91	50	519.61	110	554.74	81	3536.96	1021
2	278.60	45	142.00	41	285.58	82	1740.85	1006
4	102.37	20	73.34	40	146.78	75	532.97	224
8	65.12	21	61.40	72	125.93	174	261.55	279
16	45.92	7	59.64	104	60.92	81	150.94	441
24	53.27	28	58.34	128	61.52	55	101.33	244
32	54.87	25	54.66	123	60.61	56	97.50	468
40	52.14	29	52.87	111	64.54	127	93.17	535
Best Speedup	12.00		9.83		9.15		37.96	

* Average queue length of available tasks

Table 3: Execution times (in seconds) of 10x30 class D problems with differing # processors

more processors does lower the overall execution time, up to a point. As seen in problems 1 and 3, increasing the number of processors past a certain level (16 processors for both of these examples) does not result in further speedups, but slightly increases the execution time. This drop-off in performance gains occurs when any additional processors cease to improve pruning, and start to work on nodes not in the "critical tree" [1]. Any slight upturn in execution time is due to delay incurred while waiting for these useless evaluations to complete (this increase is bounded by the time to evaluate a single non-essential node). Even with this limit on the number of processors that could be gainfully used, all of the multiprocessor times shown here were far less than the time required for the sequential case (1 processor). Furthermore, very good speedups were observed (~38 using 40 processors for problem 4) when there was sufficient work. We are able to comment on the amount of work based on a couple of observations:

[1] It should be noted that the critical tree is dynamically determined so that at the time the processors take up tasks, it is not known whether they are in the critical tree or not.

1. The queue length for these cases is very small - in fact in smaller problem sizes we saw that it remains 1 so that adding additional processors will not help.

2. If the critical tree has n nodes, then n is a ceiling on the potential speedup achievable with inter-node parallelism, as well as a limit on the number of processors that can be gainfully used in such a parallel implementation. This limit assumes a synchronous execution model with equal node evaluation times. But even empirically (on the KSR-2) we have observed this limit to hold. Clearly, this limit does not hold for implementations that exploit intra-node parallelism.

Because of the large number of node evaluations required, the algorithm of Karabakal, Bean, and Lohmann [KBL92] could not viably solve any of 10x50 instances of class D on the KSR using only a single processor. However, our parallel implementation on the KSR is able to solve many such instances, including the five that we listed previously for our PVM implementation. The results of the runs for those five problems are given in Table 4. Please note that these times should not be directly compared with the PVM method times: the number of processors, the hardware characteristics and the algorithm characteristics are drastically different.

Problem	1	2	3	4	5
Time	2552	1197	909	1992	2716

Table 4: Execution times (in seconds) of 10x50 class D problem using 30 processors

5 Discussion

When designing and testing both the PVM and the KSR implementations the biggest factor affecting our decisions was that 85% or more of the total execution time is used in "node evaluation" (LP and knapsack for the PVM implementation, Lagrangian dual and knapsack for the KSR implementation), with the communication and support code (e.g. initialization, termination, results gathering and display) constituting less than 15% of the total time. This time breakdown implies that improvements in non-"node evaluation" code will likely have less impact on overall performance than will improvements in the node evaluation code. Viewed another way, this means that achieving a better node distribution and load balancing scheme at the cost of decreased communication and support code efficiency would likely improve the overall performance.

Our two main "performance enhancement guidelines" follow from the above observations. First, the support and communication code should first be optimized for load balancing, then for speed and efficiency. Secondly, one should make as many improvements as possible that reduce the node evaluation time. These improvements can be made by using more efficient sequential techniques (e.g. keeping columns in the LP case) and by taking advantage of further parallelism (e.g. performing the multiple independent knapsack problems for each node in parallel).

Our results for the KSR approach also indicate that simply throwing more processors at a problem will not always result in improved performance. Depending on the size and shape of the critical tree for a given problem instance, adding processors beyond a limit will result in no further increase in speed, and will actually decrease performance to some extent.

In summary, parallel execution can provide improved performance if the critical tree can be evaluated in parallel. By studying two distinct solution techniques we have found that this is indeed possible, and have been able to make the following observations:

- Execution time is dominated by node evaluations, i.e., 80%-85% of the time. Thus optimizing message passing performance, and control overhead, etc. is not a large source of performance improvement. It is also the reason why performance is not quite as sensitive to these factors.
- In a heterogeneous network (mixed speed workstations) slower processors can substantially increase execution time. Associated queue lengths are reduced at a much slower pace and these times can dominate execution time. A (simple) load balancing scheme can alleviate most of these bottlenecks.
- In the PVM version, the use of a simple centralized queue is sufficient for relatively large problem sizes and incurs relatively low overhead for queue management. In the KSR version, overhead of locking (synchronization for queue access) is minimal even up to 40 processors ($< 2\%$ in most cases).
- Algorithms should just find enough parallelism to solve the critical tree in parallel. Beyond that most of the work is spent in creating and solving nodes that are non-essential.
- Some support for state sharing (a la distributed shared memory) can substantially improve performance in the PVM version by facilitating the use of optimizations used in serial implementations.
- With low overhead queue access, and small computation/communication ratio, intra-node parallelism on the KSR was not a viable option.

6 Research Contributions

6.1 Operations Research Perspective

As shown in the PVM and KSR results sections, not only are we able to obtain substantial performance gains over the sequential algorithms, but we are able to solve instances not computationally viable with the sequential algorithms. It should be noted that these increases are obtained by parallelizing well-known sequential algorithms. This indicates that one can exploit the benefits parallelism without spending an excessive amount of time and effort in designing specialized parallel algorithms. Our approach has the additional advantage that any future sequential algorithmic improvements could likely be reflected in the parallel version as well.

6.2 Parallel Computation Perspective

Based upon the number of nodes required and the time needed to evaluate a node, we were able to characterize a branch and bound algorithm in terms of its granularity. We used this characterization to indicate which of our two platforms, PVM (coarse-grained) and KSR (medium-grained), would be best suited to each of our base algorithms. In the design of future parallel branch and bound programs, one could therefore use this guideline to make a good match between the available sequential code and the available hardware platforms. In terms of scalability, because of the number of nodes available to be executed and the relatively lower frequency of queue access, a simple central queue gave good speedups and proved to be fairly scalable for the problem sizes we considered.

Our results also suggest that a shared memory platform is more suited to branch and bound than is message-passing hardware. Aside from the "ease of programming" issue, a shared memory machine is a better match to the dynamic nature of the processors' interaction with the node queue (e.g. retrieving a node, pruning based upon a new bound).

7 Concluding Remarks

We hope that our experiences given in this report will encourage further study into the parallelism of branch and bound algorithms. In addition to our comments above, we believe that any future study work should explore the following issues:

- Can we define a precise upper bound for number of processors that achieves the potential speedup due to inter-node parallelism based on the size and shape of the critical tree?
- Could a different, possibly more complex, method of choosing the next node to evaluate result in better performance? Our current best-bound search is considered to be the overall best choice for sequential algorithms. This may not be true for parallel algorithms.
- How does the presence of multiple optimal solutions affect speedup? Is this the only reason for super-linear speedup?

References

[FJvW86] M.L. Fisher, R. Jaikumar, and L.N. van Wassenhove. A multiplier adjustment method for the generalized assignment problem. *Management Science*, 32(9):1095–1103, 1986.

[GR89] M. Guignard and M. Rosenwein. An improved dual-based algorithm for the generalized assignment problem. *Operations Research*, 37(4):658–663, 1989.

[Inc93] CPLEX Optimization Inc. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library, Version 2.1*, 1993.

[Kar92] N. Karabakal. *A C Code for Solving the Generalized Assignment Problem.* Technical Report 92-32, Department of Industrial and Operations Engineering, University of Michigan, May 1992.

[KBL92] N. Karabakal, J. Bean, and J. Lohmann. *A Steepest Decent Multiplier Adjustment Method for the Generalized Assignment Problem.* Technical Report 92-11, Department of Industrial and Operations Engineering, University of Michigan, January 1992.

[KZ88] R.M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Symposium on Theoretical Computing*, pages 290–300, 1988.

[MT90] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations.* John Wiley and Sons, 1990.

[Res92] Kendall Square Research. Technical summary, 1992.

[RS75] G.T. Ross and R.M. Soland. A branch-and-bound algorithm for the generalized assignment algorithm. *Mathematical Programming*, 8:91–103, 1975.

[Sav93] M.W.P. Savelsbergh. *A Branch-and-Price Algorithm for the Generalized Assignment Problem.* Technical Report TR COC-93-02, School of Industrial and Systems Engineering, Georgia Institute of Technology, July 1993.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.