State-of-the-Art in Parallel Search Techniques for Discrete Optimization Problems *

Ananth Grama*, and Vipin Kumar**

*Department of Computer Sciences
Purdue University, West Lafayette, IN 47907
`ayg@cs.purdue.edu`, Ph: (765) 494 6964, FAX: (765) 494 0739

**Department of Computer Science
University of Minnesota, Minneapolis, MN 55455
`kumar@cs.umn.edu`, Ph: (612) 624 8023, FAX: (612) 625 0572

Abstract

Discrete optimization problems arise in a variety of domains such as VLSI design, transportation, scheduling and management, and design optimization. Very often, these problems are solved using state space search techniques. Due to the high computational requirements and inherent parallel nature of search techniques, there has been a great deal of interest in the development of parallel search methods since the dawn of parallel computing. Significant advances have been made in the use of powerful heuristics and parallel processing to solve large scale discrete optimization problems. Problem instances that were considered computationally intractable only a few years back are routinely solved currently on server-class symmetric multi-processors and small workstation clusters. Parallel game-playing programs are challenging the best human minds at games like chess. In this paper, we describe the state of the art in parallel algorithms used for solving discrete optimization problems. We address heuristic and non-heuristic techniques for searching graphs as well as trees, and speedup anomalies in parallel search that are caused by the inherent speculative nature of search techniques.

*This work is sponsored by the by Army Research Office contract DA/DAAG55-98-1-0441, NSF Grant EIA9876014, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

1 Introduction

The generalized discrete optimization problem (DOP) deals with minimizing an objective function of a set of variables that are allowed only discrete values. An instance of this class is the integer programming problem in which only integer solutions are admissible for a linear programming (LP) model. Other instances arise in diverse domains such as VLSI design (placement of VLSI components on a die to minimize silicon area), robotics (robot motion planning), and logistics (visiting all vertices of a graph while minimizing edge cost). Most such problems are NP complete; hence their solution time increases exponentially in the size of the problem for all known algorithms. One may argue that it is pointless to apply parallel processing to these problems since we can never reduce their worst-case run time to a polynomial without using an exponential number of processors. However, the average-time complexity of heuristic search algorithms for some problems is polynomial [38, 46]. Furthermore, there are heuristic search algorithms that find suboptimal solutions for specific problems in polynomial time. In such cases, bigger problem instances can be solved using parallel computers. Many DOPs (such as robot motion planning, speech understanding, and task scheduling) require real-time solutions. For these applications, parallel processing may be the only way to obtain acceptable performance. Problems for which optimal solutions are highly desirable can sometimes be solved for moderate sized instances in a reasonable amount of time by using parallel search techniques (for example, VLSI floor-plan optimization).

Due to the high computational requirement and inherent parallel nature of search techniques, there has been great interest in the development of parallel search methods since the dawn of parallel computing. In fact, parallel search programs were among the first non-numerical programs written for early generation parallel machines such as HEP and C.mmp. There are many characteristics of search techniques that make their parallel formulations very different from those of traditional numerical algorithms, but much closer to other non-numerical algorithms. Parallel search algorithms rely on dynamic data structures, which makes it difficult to use static work distribution techniques. There is potential for speculative work both in serial and parallel search algorithms. Hence load balancing algorithms for parallel search must consider qualitative information as well. Some search algorithms require use of large global hash structures, which can be hard to implement on losely coupled parallel architectures. Tremendous advances have been made both in theoretical and applied aspects of parallel search techniques [11]. These techniques have been used to solve problems that are well beyond the scope of conventional serial processors. In this paper, we provide a brief overview of commonly used sequential search methods, and present the state-of-the-art in the parallel formulations of various search techniques.

2 Overview of Search Techniques

Very often a discrete optimization problem is formulated as the problem of finding a path in a graph from a designated initial node to one of several possible goals [38, 20]. Such a graph is called a state space. In some problems, any solution path between the initial node and a goal node is acceptable, whereas in other problems a cost is associated with each path and a minimum cost solution is desired. In many applications, it is possible to compute a lower bound on the cost of a solution path that passes through a given state. This cost is also called a heuristic estimate.

Two commonly used techniques for searching state spaces are depth-first and best-first search. Depth-first search techniques start from an initial state and generate a set of successor states. One of these most recently generated states is then selected for expansion at each step. Many variants of this simple DFS technique exist. In one such variant, the successor states of a node are ordered on the basis of their likelihood of yielding a solution. This search technique is referred to as directed-DFS or ordered backtracking. In many applications, early branches explored by DFS techniques may lead to large subtrees containing no solutions when solutions may be available close to the root on alternate branches. In these cases, it is useful to limit the depth in terms of cost or number of edges, to which exhaustive search is performed. If no desirable solution is found in this limited search, the limit is revised and search performed again. The limit can be defined in terms of tree depth (depth-bounded DFS), or in terms of solution cost (IDA*). In depth-first search, if the lower bound of a node is larger than the cost of the best solution found thus far, then the node can be pruned. This technique is referred to as depth-first branch-and-bound (DFBB).

In all algorithms based on the DFS technique, once a successor node n is selected for expansion, its sibling nodes are expanded only after the entire tree rooted at node n is expanded, even if successors of n are much less promising than the siblings of n. Thus, these algorithms make use of the heuristic information only on a local scale to order the successors of a node. DFBB and IDA* also make a limited use of heuristic information on a global scale as they prune a node if it is less promising than a previously found solution or bound. As we will discuss later, this limited use of heuristics makes it easy to develop parallel formulations of DFS algorithms.

In contrast to DFS, best-first search algorithms expand the most promising node currently available on the search frontier, and thus make complete use of the available heuristic information. The search frontier is maintained as a priority queue, and is commonly referred to as the OPEN list[1]. A well known algorithm of this type is A*, also referred to as best-first branch and bound (BFBB). A major price paid for this use of heuristic information in a global context is that BFS requires memory proportional to the size of the search frontier, which is often exponential in the depth of the solution. In contrast, all DFS algorithms require memory proportional to the depth of the solution path.

In many applications, identical states can be reached from multiple paths. Re-expansion of these states can be avoided if the search technique checks every generated node for possible replications. BFS strategies support replication checking by maintaining a list of all the expanded nodes (and their costs) on a list called CLOSED[1]. DFS based techniques do not support replication checking, and thus essentially unroll the graph into a tree. Unrolling graphs into trees can have overheads ranging from a constant factor to exponential depending on the structure of the graph. In the latter case, DFS techniques may be unsuitable, as they will need to expand many more nodes than BFS. Finally, simple and directed DFS are generally used to find the first solution that they encounter in the search space, whereas DFBB, IDA*, and BFS find a least cost solution. This taxonomy of search characteristics is illustrated in Figure 1.

[1] OPEN and CLOSED data structures are called lists for historical reasons. In fact, OPEN list is typically implemented using a heap and CLOSED list is implemented as a hash table.

Although a central component of most parallel search algorithms is a dynamic load balancing scheme, the parallel formulation is impacted by the nature of the search space, the use of heuristics, memory requirements, and the desired solution quality. In the following sections, we discuss the parallel formulations developed for various search algorithms.

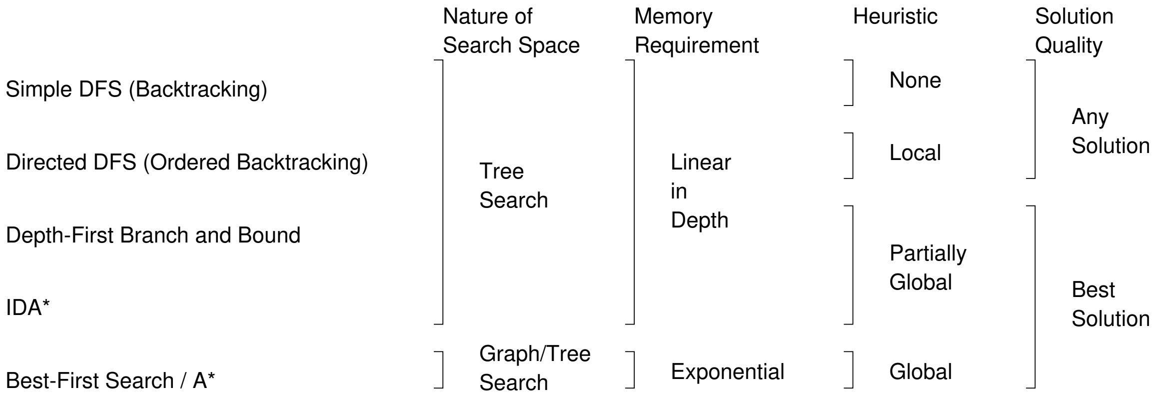

Figure 1: Various enumerative search procedures and their characteristics.

3 Depth-First Search

Depth-first search is particularly amenable to parallel processing primarily because a subtree rooted at any node can be searched relatively independently of the search at other nodes or subtrees. In fact, searching different subtrees concurrently requires either no communication (e.g., backtracking, IDA*) or minimal communication (e.g., DFBB). This is due to the fact that DFS techniques do not make use of heuristics on a global scale. It would therefore seem that by statically assigning nodes in a tree to processors, it is possible to derive parallel formulations. However, for most applications, trees generated by DFS tend to be highly irregular, and it is very difficult to estimate the work associated with a node a-priori. Hence, any static allocation of subtrees to processors is bound to result in significant load imbalance among processors. The core of parallel formulations of DFS algorithms is thus a dynamic load balancing technique that minimizes inter-processor communication and processor idling.

A number of load distribution techniques have been developed for parallel DFS [25, 41, 22, 17, 21]. Load balancing techniques may be receiver initiated or sender initiated. In receiver initiated techniques, when a processor becomes idle, it requests a selected processor for work. Many different selection policies have been proposed such as use of a centralized server, random polling [12], nearest neighbor [25], global round-robin [25], and asynchronous round robin [12, 25]. In contrast to receiver initiated load balancing, in sender initiated schemes, a processor sends work to selected processors as it is generated. Selection strategies for sender initiated work transfers include hierarchical techniques (with super-servers, servers and clients) [13], random sampling [25], and near-neighbor techniques [41]. The quanta of work farmed out to processors in sender and receiver initiated schemes ranges from a single state (node in the tree) to forests of subtrees resulting from stack splitting [12, 25]. Schemes for fault tolerance using acknowledgements have also been built into load balancing techniques [12].

The performance and scalability of these techniques is often dependent on the underlying architecture. Many of these techniques are, in principle scalable, i.e., they result in linear speedup on increasing the number of processors p as long as the size of the search space grows fast enough with p. It is desirable that this required rate of growth of problem size (also referred to as the Isoefficiency metric [24]) be as small as possible since it allows the use of a larger number of processors effectively for solving a given problem instance.

The irregular and dynamic nature of the search space makes it difficult to precisely compute the performance and scalability of parallel DFS. However, for trees with branching factor greater than $1 + \epsilon$ (for a small constant ϵ), it is often possible to partition the computation associated with a collection of nodes into two parts such that the smaller of these two parts can be guaranteed to have a minimum fixed constant fraction of the computation associated with the original node. This property has been used to derive parallel formulations of DFS with isoefficiency of $O(p \log^2 p)$ on hypercube and $O(p^{1.5} \log p)$ on mesh connected parallel computers [25].

The parallel formulation of DFS described above can be applied to DFBB with one minor modification. In DFBB, we need to keep all the processors informed of the current best solution. On a shared address space architecture, this can be done easily by maintaining a global best solution in globally accessible memory. On a message passing computer, this can be done by allowing each processor to maintain the current best solution known to it. Whenever a processor finds a solution path better than the current best known, it broadcasts it to all the other processors which update (if necessary) their current best solution path. Note that if a processor's current best solution path is worse than the global best solution path, then it only affects the efficiency of the search but not its correctness. The overhead for maintaining the current best solution tends to be a small fraction of the overhead for dynamic load balancing. Parallel formulations of DFBB have been shown to yield near linear speedups for many problems and architectures [45, 27, 8, 7, 1].

Termination of Parallel Search

If only one solution is desired to a problem instance being solved using depth-first search, then as soon as any processor finds a solution, search at other processors needs to be terminated. On some parallel architectures, it is possible for one processor to interrupt other processors and send them a special signal. If such a mechanism is not available, then each processor needs to perform a periodic check to see if a solution has been found.

If all the solutions (or all best solutions) to a problem instance are desired, then parallel DFS searches a well defined portion of the entire search tree (determined by the specific algorithm and heuristic used). Since this search space is dynamically allocated to processors, it is not sufficient to poll each processor serially to check if they have exhausted their work. The reason is that a processor may pick up work after it has been polled. Hence a distributed termination detection algorithm is needed to ensure that all processors have indeed exhausted the search space [5].

4 Best-First Search

Best-first search techniques are more difficult to parallelize for the following two reasons: (i) they use heuristics on a global scale, making it harder to utilize the full power of heuristics in a parallel setting; and (ii) they need to maintain the CLOSED list if the search space is a graph. We first discuss parallel formulations of best-first search in the context of state-space trees, and subsequently discuss the case for graphs.

BFS methods may appear inherently serial since they require expansion of the most promising node on the OPEN list at each step. However, it is possible that many nodes on the OPEN list are equally promising, in which case, they can all be expanded concurrently. If the number of such (equally) promising nodes is fewer than the number of available processors, then the other less promising nodes available on OPEN list can be searched by the idle processors. These less promising nodes may not be searched by serial BFS, and thus form speculative work. The effective degree of concurrency of such a formulation is essentially equal to the average number of nodes available on the OPEN list such that their cost is less than that of the optimal solution. Since the cost of the optimal solution is unknown, a good strategy is to always expand only the p-most promising nodes currently available on the OPEN list; where p is the number of processors. As long as p is no more than the effective degree of concurrency, all processors will tend to search nodes that will also be searched by serial BFS. The effective degree of concurrency is usually dependent on the size of the search space (larger spaces have higher degree of concurrency) and the heuristic function (stronger heuristics lead to smaller degree of concurrency).

The basic data structure of BFS, the OPEN list, is a priority queue that must support removal of the best entry in the queue and insertion of nodes into the queue. The data structure most often used for this is a heap. The simplest parallel formulation of BFS uses multiple processors working on a heap stored in globally addressable memory. Each processor locks the heap and picks out the current best node. The heap is unlocked and the node is expanded to yield its children. These children are reinserted into the heap with appropriate locking. The use of a global heap is a source of contention. If the time taken to lock, remove, and unlock the top element of the heap is t_{access} and time for expansion is t_{exp}, then the speedup of the formulation is bounded by $(t_{access} + t_{exp})/t_{access}$.

Many techniques have been developed to effectively reduce t_{access} and thus increase the available parallelism. These techniques support multiple operations on heaps stored in shared memory while maintaining the strict deletion and insertion ordering [42]. The performance of these schemes scales to dozens of processors but is ultimately limited by the locking and access times as specified above.

Another way of alleviating the contention overhead associated with a single global heap is to use multiple heaps. The concept of p processors sharing a single heap can be relaxed to k processors sharing a heap with p/k heaps in all. The number k is selected by studying the overheads associated with communication and locking, contention, and node expansion time. In the extreme case, each processor has its own heap. A simple parallel formulation based on multiple heaps starts with the initial state in a single heap. As additional nodes are generated, they are farmed out to other heaps (using schemes similar to load balancing in DFS). Once all the heaps have been populated, processors can perform BFS on individual heaps. This formulation has a significant drawback in that although it balances computation among processors, it does not guarantee the quality of nodes individual processors explore. Consequently, processors may spend considerable time exploring parts of the search space that may not be explored by the serial formulation.

To alleviate this drawback of excess work performed by the parallel formulation, one must ensure that all heaps have a share of the most promising nodes globally available. This process, also referred to as quality equalization, ensures that the best nodes available globally are evenly distributed across the heaps. Since the quality of the nodes changes as computation proceeds, quality equalization must be performed periodically. A variety of triggers and mechanisms have been used for quality equalization [30, 6, 45, 4]. A simple triggering mechanism tracks the best node in the system. The best node in the local heap is compared to the best node in the system and if it is considerably worse, an equalization process is initiated. Alternately, an equalization process may be initiated periodically.

Quality equalization requires movement of nodes between heaps. This movement of nodes can be structured (into a ring, or via a shared blackboard) or randomized. In randomized equalization, a processor periodically selects a random heap, picks up the best node from the heap, and inserts it into the local heap. Alternately, heaps can be organized into hierarchical structures. At the lower levels, equalization is performed more frequently than the higher levels in the hierarchy. Several triggering mechanisms and redistribution strategies have been explored [6, 47]. It has been shown that these schemes are capable of providing linearly increasing speedups with number of processors if the problem size is increased appropriately. Specifically, scalable parallel formulations of best-first tree search with isoefficiency of $O(p \log^2 p)$ on hypercube architectures have been developed [30]. Using these schemes, speedups in excess of 950 have been demonstrated on 1024 processor hypercubes in the context of TSPs formulated as best-first tree search problems [6].

Best-First Search of State Space Graphs

Efficient parallel implementation of BFS on state space graphs requires a distributed mechanism that allows many processors to perform duplicate checking or insertions into the CLOSED list concurrently. The nature of implementation of this mechanism is dependent on the underlying parallel architecture.

In shared address space machines, insertion of nodes into the closed list requires locking of the list. If there is a single lock associated with the entire list, the list must be locked approximately as many times as the total number of nodes expanded. This represents a serial bottleneck. The bottleneck can be alleviated by associating multiple locks with the closed list. Processors lock only relevant parts of the closed list into which the node is being inserted. Message passing formulations of this parallel search technique distribute the closed list across the processors. Each processor expands a node and hashes the successors to appropriate processors via a message. The destination processor checks for replication and sends a message back specifying whether the node exists in the closed list or not. Such a protocol requires two-way cooperation between various processors. Each processor must periodically check for incoming messages, process them and send appropriate responses.

A simpler parallel formulation uses the hash function both for replication checking and for qualitative and quantitative load balance. In this formulation, the successors of a node are hashed to appropriate processors. The destination processor checks for replication as before. However, instead of sending a response back, it inserts the successor nodes into its own open list. This has two main advantages: (i) it eliminates the need for a return message and wait at the source processor; and (ii) the randomized hash function serves as the method for balancing load qualitatively and quantitatively. These schemes have been studied by many researchers [33, 30]. Assuming a perfectly random hash function, it has been shown that if the number of nodes originating at each processor grows as $O(\log p)$, then each processor will have asymptotically equal number of nodes after the hash operation [33].

One of the major drawbacks of graph search techniques such as BFS is that their memory requirement grows linearly with the search space. For large problems, this memory requirement becomes prohibitive. Many limited-memory variants of heuristic search have been developed. These techniques rely on retraction or delayed expansion of less promising nodes to reduce memory requirement. In the parallel processing context, retractions lead to additional communication and indexing for parent-child relationships [9].

5 Game Tree Search

Min-max techniques such as $\alpha - \beta$ game tree search are extensively used in game-playing programs. In this algorithm, the minimax value of a node is determined for a given search window $[\alpha, \beta]$. This window serves to prune significant parts of the tree that do not effect the minimax evaluation of the root. Initially, this window is set to $[-\infty, \infty]$. The first child of a node is searched with the same window as the parent. The window of other children is narrowed on the basis of results from previously visited children. This process is continued until a specified depth is reached, or the branch has no successors. Variants of this algorithm are based on iterative deepening, which guides search based on results from previous iterations.

Min-max techniques present considerable challenges for parallel processing. Conceptually, $\alpha - \beta$ game tree search algorithm can also be viewed as a depth-first branch-and-bound algorithm [23] that searches for a maximum payoff strategy among all strategies represented in the game tree. However, game trees tend to be strongly ordered. Therefore, naive parallel formulations may expand a large number of nodes that are not expanded by the serial formulation. Second, the information about the current best strategy cannot be captured by a single node, as the aim is to find a best winning strategy rather than a winning position. Effective propagation of this information to other processors searching disjoint parts of the search space is much harder compared to sharing of the cost of current best solution in traditional DFBB algorithms. A number of researchers have investigated parallel formulations of $\alpha - \beta$ [10, 34, 40]. The problem of excess node expansions is handled by using a prioritized work distribution in which the tree is expanded left to right in a sweeping pattern also referred to as the “Young Brothers Wait Concept”. Using these optimizations along with dynamic load balancing, researchers have demonstrated speedups of as much as two orders of magnitude on a variety of parallel platforms.

The utility of parallel processing has been demonstrated in the context of a number of games, and in particular, Chess. Work on large scale parallel $\alpha - \beta$ search led to the development of deep thought [15] in 1990. This program was capable of playing chess at grandmaster level. Subsequent advances in the use of dedicated hardware, parallel processing, and algorithms resulted in the development of IBM's Deep Blue [16, 14] that beat the reigning world champion Gary Kasparov. Feldmann et al. [40] developed a distributed chess program that is acknowledged to be one of the best computer chess players based entirely on general purpose hardware.

6 Speedup Anomalies in Parallel Formulations of Search Algorithms

In parallel search algorithms, the speedup can differ greatly from one execution to another. This is because the portions of the search space examined by different processors are determined dynamically and can be different for different executions. A number of researchers have observed and analyzed the existence of these speedup anomalies. An acceleration anomaly manifests itself in the form of a speedup greater than p on p processors when parallel search does less work than the serial search algorithm [26, 29]. A speedup of less than p, attributed to excess work done by the parallel formulation compared to the sequential formulation, is termed as a deceleration anomaly [26, 28]. The ratio W_p/W_s, where W_p and W_s are the number of nodes searched by parallel and serial formulations, is called the search overhead factor.

Speedup anomalies can easily occur in parallel depth-first search for finding any solution, as certain execution sequences may find a solution much earlier than others. However, it appears natural that, on the average, the search overhead factor for parallel DFS should always be greater than or equal to one. Otherwise the serial search algorithm is not optimal; i.e., the emulation of parallel DFS on a single processor will be faster than sequential DFS. Surprisingly, there are situations, in which parallel DFS can have a search overhead factor of less than 1 on the average, implying that the serial search algorithm in the situation is suboptimal. Kumar and Rao [43] show that if no heuristic information is available to order the successors of a node, then on the average, the speedup obtained by parallel DFS is superlinear if the distribution of solutions is non-uniform. This model is validated by experiments on synthetic state-space trees modeling the hackers problem, the 15-puzzle problem and the n-Queens problem. In all these experiments, serial and parallel DFS do not use any heuristic ordering, and select successors arbitrarily. The basic reason for this phenomenon is that parallel search can invest resources into multiple regions of the search frontier concurrently. When the solution density in different regions of the search frontier is non-uniform and these non-uniformities are not known a priori, then sequential search has an equal chance of searching a low density region (and thus be forced to expand many nodes) or a high density region. On the contrary, parallel search can search all regions at the same time, ensuring that at least some processors are searching the high density regions.

If good heuristics are available to order the successors of nodes in DFS, then the search overhead factor of parallel DFS may be greater than one. The reason is that some processors may search unpromising areas of the search space. This phenomenon is most visible in the context of $\alpha - \beta$ search of game trees for which strong ordering heuristics are available. Kale and Saletore [19] present a scheme that combines the information about the ordering and depth of nodes to perform load balancing among processors. This scheme largely maintains the depth-first heuristic order of search across processors, and results in consistent linear speedup. Even in the presence of good ordering heuristics, parallel DFS may lead to superlinear speedup. For example, this can happen if there is a non-zero probability that the heuristic can guide search to large regions that do not contain any solutions [43].

In parallel BFS, the search overhead can be greater than one if some processors expand nodes that are never expanded by serial BFS (e.g., nodes that have higher bound than the cost of the optimal solution). In fact, it can be shown that for any given BFS instance, there exists a constant k such that expanding more than k nodes in parallel from a global open list leads to wasted computation [26]. This situation gets worse with distributed open lists since expanded nodes have locally minimum heuristics that are not the best nodes across all open lists.

The search overhead for parallel BFS can also be less than one. This can happen when multiple nodes on the open list have identical heuristic values, but require different amount of search to detect a solution (or to find out that the node does not lead to optimal solution). Hence, if a parallel algorithm happens to explore the better nodes first, then it can result in acceleration anomalies, and vice versa. It is possible to derive properties of the heuristic function such that superlinearity effects in parallel BFS are made deterministic [29, 32].

7 Role of Heuristics

Heuristics form the most important component of search techniques, and parallel formulations of search algorithms must be viewed in the context of these heuristics. In BFS techniques, heuristics focus search by lowering the effective branching factor. In DFBB methods, heuristics provide better bounds, and thus serve to prune the search space.

Often, there is a tradeoff between the strength of the heuristic and the effective size of search space. Better heuristics result in smaller search spaces but are also more expensive to compute. For example, an important application of strong heuristics is in the computation of bounds for mixed integer programming (MIP). Mixed integer programming has seen significant advances over the years [18]. Whereas fifteen years back, MIP problems with 100 integer variables were considered challenging, today, many problems with up to 1000 integer variables can be solved on workstation class machines using branch-and-cut methods. Largest known instances of TSPs and QAPs have been solved using branch-and-bound with powerful heuristics [3, 35]. The presence of effective heuristics may prune the search space considerably. For example, when Padberg and Rinaldi introduced the branch-and-cut algorithm in 1987, they used it to solve a 532 city TSP, which was the largest TSP solved optimally at that time. Subsequent improvements to the method led to the solution of a a 2,392 city problem [36]. More recently, using cutting planes, problems with over 7,000 cities have been solved [18] on serial machines. However, for many problems of interest, the reduced search space still requires the use of parallelism [3, 35, 44, 31]. Use of powerful heuristics combined with effective parallel processing has enabled the solution of extremely large problems [35]. For example, QAP problems from the Nugent and Eschermann test suites with up to 4.8×10^{10} nodes (Nugent22 with Gilmore-Lawler bound) were solved on a NEC Cenju-3 parallel computer in under nine days [3]. Another dazzling demonstration of this was presented by the IBM Deep Blue. Deep blue used a combination of dedicated hardware for generating and evaluating board positions and parallel search of the game tree using an IBM SP2 to beat the current world chess champion, Garry Kasparov [16, 14].

Heuristics have several important implications for exploiting parallelism. Strong heuristics narrow the state space and thus reduce the concurrency available in the search space. Use of powerful heuristics pose other computational challenges for parallel processing as well. For example, in branch-and-cut methods, a cut generated at a certain state may be required by other states. Therefore, in addition to balancing load, the parallel branch-and-cut formulation must also partition cuts among processors so that processors working in certain LP domains have access to the desired cuts [2, 27, 8].

In addition to inter-node parallelism that has been discussed until this point, intra-node parallelism can become a viable option if the heuristic is computationally expensive. For example, the assignment heuristic of Pekny et al. has been effectively parallelized for solving large instances of TSPs [35]. If the degree of inter-node parallelism is small, this form of parallelism provides a desirable alternative. Another example of this is in the solution of MIP problems using branch-and-cut methods. Branch-and-cut methods rely on LP relaxation for generating lower bounds of partially instantiated solutions followed by generation of valid inequalities [18]. These LP relaxations constitute a major part of the overall computation time. Many of the industrial codes rely on Simplex to solve the LP problem since it can adapt the solution to added rows and columns. While interior point methods are better suited to parallelism, they tend to be less efficient for reoptimizing LP solutions with added rows and columns (in branch-and-cut methods). LP relaxation using Simplex has been shown to parallelize well on small numbers of processors but efforts to scale to larger numbers of processors have not been successful. LP based branch and bound methods have also been used for solving quadratic assignment problems using iterative solvers such as preconditioned Conjugate Gradient to approximately compute the interior point directions [37]. These methods have been used to compute lower bounds using linear programs with over 150,000 constraints and 300,000 variables for solving QAPs. These and other iterative solvers parallelize very effectively to a large number of processors. A general parallel framework for computing heuristic solutions to problems is presented by Pramanick and Kuhl [39].

8 Concluding Remarks

Great advances have been made in the development of scalable parallel formulations of various search methods in the past two decades. Performance and scalability of load balancing techniques for parallel depth-first search methods is largely understood, and it is unlikely that any new methods will perform significantly better than the existing methods. Parallel best-first search methods require qualitative load balancing in addition to quantitative load balancing used in depth-first search, and are thus harder to analyze. However, many methods for qualitative load balancing have been developed and shown to have provably good performance under a reasonable set of assumptions. For a wide variety of problems, both best-first and depth-first search methods have shown excellent performance and scalability up to 1024 processors [45, 25, 1, 40, 30, 44]. Attention has now shifted to the development of software environments that allow practitioners to solve real problems on parallel machines [4].

It is important to note that parallel processing can only provide speed up proportional to the number of processors. Hence, parallel search can be used to solve significantly bigger instances of the problem only if the search space grows as a small exponent of problem size. Hence, challenges remain in discovering better heuristics that reduce the growth of the search space. Indeed, for many problems, gains due to new powerful heuristics may be much higher than those due to massively parallel formulations using existing heuristics. These powerful heuristics may also offer new opportunities for exploiting parallelism.

References

[1] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. Floorplan optimization on multiprocessors. In *Proceedings of the 1989 International Conference on Computer Design*, 1989.

[2] R. E. Bixby, W. Cook, A. Cox, and E. Lee. Parallel mixed integer programming. Technical Report CRPC TR 95554, Center for Research on Parallel Computation, Research Monograph, 1995.

[3] Adrian Brungger, Ambros Marzetta, Jens Clausen, and Michael Perregaard. Solving large-scale qap problems in parallel with the search library zram. *Journal of Parallel and Distributed Computing*, 50:157–169, 1998.

[4] B. L. Cun and C. Roucairol. BOB: A unified platform for implementing branch-and-bound like algorithms. Technical Report N95/16, Universiti de Versailles Saint Quentin, 1995. Available from `http://www.prism.uvsq.fr/english/parallel/cr/bob_us.html`.

[5] E. W. Dijkstra, W. H. Seijen, and A. J. M. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16-5:217–219, 1983.

[6] S. Dutt and N. R. Mahapatra. Scalable load-balancing strategies for parallel a* algorithms. *Journal of Parallel and Distributed Computing*, 22(3):488–505, Sept. 1994. Special Issue on Scalability of Parallel Algorithms and Architectures.

[7] Jonathan Eckstein. Parallel branch-and-bound methods for mixed-integer programming on the cm-5. *SIAM Journal on Optimization*, 4(4):794–814, 1994.

[8] Jonathan Eckstein. Distributed versus centralized storage and control for parallel branch and bound: Mixed integer programming on the cm-5. *Computational Optimization and Applications*, 7(2):199–220, 1997.

[9] M. Evett, James Hendler, Ambujashka Mahanti, and Dana Nau. PRA*: A memory-limited heuristic search procedure for the connection machine. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 145–149, 1990.

[10] Chris Ferguson and Richard Korf. Distributed tree search and its application to alpha-beta pruning. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, August 1988.

[11] A. Ferreira and P. Pardalos, editors. *Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques*, volume 1054 of *LNCS State-of-the-Art Surveys*. Springer-Verlag, 1996.

[12] Raphael A. Finkel and Udi Manber. DIB - a distributed implementation of backtracking. *ACM Transactions of Programming Languages and Systems*, 9 No. 2:235–256, April 1987.

[13] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for OR-parallel exhaustive search programs on the Multi-PSI. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 50–59, 1990.

[14] Scott Hamilton and Lee Garber. Deep Blue's hardware-software synergy. *IEEE Computer*, 30(10):29–35, October 1997.

[15] F.-H. Hsu. Large scale parallelization of alpha-beta search: An algorithmic and architectural study with computer chess. Technical report, Carnegie Mellon University, Pittsburgh, PA, 1990. Ph.D. Thesis.

[16] F.-H. Hsu, M. S. Campbell, and A. J. Hoane. Deep Blue system overview. In *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain*, pages 240–244, 1995.

[17] Virendra K. Janakiram, Dharma P. Agrawal, and Ram Mehrotra. A randomized parallel backtracking algorithm. *IEEE Transactions on Computers*, C-37 (12), 1988.

[18] L. Johnson, G. Nemhauser, and M. Savelsbergh. Progress in integer programming: An exposition. Technical report, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1997. Available from `http://akula.isye.gatech.edu/ mwps/mwps.html`.

[19] L. V. Kale and V. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *International Journal of Parallel Programming*, 3, August 1990.

[20] L. N. Kanal and Vipin Kumar. *Search in Artificial Intelligence*. Springer-Verlag, New York, NY, 1988.

[21] R. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of ACM*, 40:765–789, 1993.

[22] George Karypis and Vipin Kumar. Unstructured Tree Search on SIMD Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(10), October 1994.

[23] V. Kumar and L. N. Kanal. A general branch-and-bound formulations for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.

[24] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley (ISBN 0-8053-3170-0), Redwod City, 1994.

[25] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994.

[26] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. *Communications of the ACM*, pages 594–602, 1984.

[27] Eva K. Lee and John E. Mitchell. Computational experience of an interior-point algorithm in a parallel branch-and-cut framework. In *Proceedings for SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[28] G. J. Li and B. W. Wah. Computational efficiency of combinatorial or-tree searches. *IEEE Trans. on Software Engineering*, 16(1):13–31, Jan. 1990.

[29] Guo-Jie Li and Benjamin W. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, C–35, June 1986.

[30] N. R. Mahapatra and S. Dutt. Scalable global and local hashing strategies for duplicate pruning in parallel a* graph search. *IEEE Trans. Parallel and Distributed Systems*, 8(7), July 1997.

[31] B. Mans, T. Mautor, and C. Roucairol. A parallel depth first search branch and bound for the quadratic assignment problem. *European Journal of Operational Research*, 81(3):617–628, 1995.

[32] B. Mans and C. Roucairol. Performances of parallel branch and bound algorithms with best-first search. *Discrete Applied Mathematics*, 66(1):57–76, April 1996.

[33] G. Manzini and M. Somalvico. Probabilistic performance analysis of heuristic search using parallel hash tables. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, 1990.

[34] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14:533–551, 1982.

[35] D. L. Miller and J. F. Pekny. The role of performance metrics for parallel mathematical programming algorithms. *ORSA Journal on Computing*, 5(1), 1993.

[36] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, 1991.

[37] P. Pardalos, Y. Li, K.G. Ramakrishna, and M.G.C. Resende. Lower bounds for the quadratic assignment problem. *Annals of Operations Research*, 50:387–411, 1994. Special Volume on Applications of Combinatorial Optimization.

[38] Judea Pearl. *Heuristics-Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.

[39] Ira Pramanick and Jon G. Kuhl. An inherently parallel method for heuristic problem-solving: Part i-general framework. *IEEE Transactions on Parallel and Distributed Systems*, 6(10), October 1995.

[40] B. Monien R. Feldmann, P. Mysliwietz. Studying overheads in massively parallel min/max-tree evaluation. In *Proc. of the 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, 1994.

[41] A. G. Ranade. Optimal speedup for backtrack search on a butterfly network. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[42] V. Nageshwara Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, C-37(12), 1988.

[43] V. Nageshwara Rao and Vipin Kumar. On the efficicency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993. Also available as Technical Report TR 90-55, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[44] C. Roucairol. A parallel branch and bound algorithm for the quadratic assignment problem. *Discr. Appl. Math.*, 18:211–225, 1987.

[45] B. Monien S. Tschvke, R. L|ling. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *Proc. of the 9th International Parallel Processing Symposium*, pages 182–189, Santa Barbara, CA, April 1995.

[46] Benjamin W. Wah, Gou-Jie Li, and C. F. Yu. Multiprocessing of combinatorial search problems. *IEEE Computer*, June 1985.

[47] Benjamin W. Wah and Y. W. Eva Ma. MANIP—a multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, c–33, May 1984.