

## The parallel search bench ZRAM and its applications

Adrian Brünger<sup>a</sup>, Ambros Marzetta<sup>b</sup>, Komei Fukuda<sup>c</sup> and Jurg Nievergelt<sup>d</sup>

*<sup>a</sup>Novartis Pharma AG, CH-4002 Basel, Switzerland*

E-mail: adrian.bruengger@pharma.novartis.com

*<sup>b</sup>The International Computer Science Institute, Berkeley, CA 94704, USA*

E-mail: ambros.marzetta@iaeth.ch

*<sup>c</sup>Institute of Operations Research, Swiss Federal Institute of Technology,  
CH-8092 Zürich, Switzerland*

E-mail: fukuda@ifor.math.ethz.ch

*<sup>d</sup>Institute of Theoretical Computer Science, Swiss Federal Institute of Technology,  
CH-8092 Zürich, Switzerland*

E-mail: nievergelt@inf.ethz.ch

Distributed and parallel computation is, on the one hand, the cheapest way to increase raw computing power. Turning parallelism into a useful tool for solving new problems, on the other hand, presents formidable challenges to computer science. We believe that parallel computation will spread among general users mostly through the ready availability of convenient and powerful program libraries. In contrast to general-purpose languages, a program library is specialized towards a well-defined class of problems and algorithms. This narrow focus permits developers to optimize algorithms, once and for all, for parallel computers of a variety of common architectures. This paper presents ZRAM, a portable parallel library of exhaustive search algorithms, as a case study that proves the feasibility of achieving simultaneously the goals of portability, efficiency, and convenience of use. Examples of massive computations successfully performed with the help of ZRAM illustrate its capabilities and use.

### 1. The role of computing power for combinatorial search

For half a century since computers came into existence, the goal of finding elegant and efficient algorithms to solve “simple” (well-defined and well-structured) problems has dominated algorithm design. Over the same time period, both processing and storage capacity of computers have increased by roughly a factor of  $10^6$ . The next few decades may give us a similar rate of growth in raw computing power, due to various factors such as continuing miniaturization, parallel and distributed computing.

If a quantitative change of orders of magnitude leads to qualitative changes, where will the latter take place? Many discrete combinatorial problems exhibit no detectable regular structure to be exploited, they appear “chaotic”, and do not yield to efficient algorithms. For such problems, exhaustive search of large state spaces appears to be the only viable approach [23].

Whereas the asymptotic complexity of polynomial-time algorithms provides an excellent basis for estimating computing times, meaningful a priori estimates are more difficult to obtain for problems that require exhaustive search.

The latter exhibit many differences from those that admit efficient algorithms. Typically, they are NP-hard, a technical term that has often been interpreted as “computationally intractable”. But we consider this interpretation to be misleading. It is rarely the case that all problem instances in such a class are computationally demanding; it is not even necessarily the case that the average instance is hard – it is merely certain that there exist computationally intractable instances in such a class. But such hard instances might be sparse or, more to the point, not of a type that tends to occur in applications. The (Euclidean) Traveling Salesman Problem is an example where instances have been solved that are much larger than earlier theoretical insights would have led one to expect.

Because of the great variation in complexity between different instances of the same problem class, even among those of equal input size, the problems to be tackled are often individual instances, rather than a class parametrized by input size  $n$ . For example, we have solved Sam Loyd’s 15-puzzle played on a  $4 \times 4$  array [6]. From this instance, with a state space of size  $10^{13}$ , nothing of great interest follows for a generalized sliding puzzle played on an  $n \times n$  array.

If asymptotics, the most powerful technique for performance estimation of efficient algorithms, does not help, we are reduced to empirical observations. One rule of thumb states that several state spaces of irregular structure have been exhaustively searched up to a size of about  $10^{10}$ . This observation does not give us much information about the chances of solving the 15-puzzle, whose state space is 1000 times larger. It is difficult to predict, for a given instance of an exhaustive search problem, whether it can be solved in an hour, or a year, or at all.

If we cannot predict what computational resources are required, there is always the hope that a computer several orders of magnitude faster may do the job. Indeed, experimentation with faster computers to explore the limits of computational power has always driven scientific computing. Most of it was of the numeric type. There is much less experience with discrete, combinatorial computing, despite impressive early achievements such as the proof of the Four Color Theorem [1].

Insight into the power of exhaustive search for solving unstructured combinatorial problems can only come from empirical tests – problems that challenge today’s computational resources. Such experimentation calls for a combination of four ingredients and that is the subject to this paper: suitable benchmark problems, general-purpose search algorithms, convenient software tools, and powerful computers.

Parallel and distributed computing have made vast raw computational power widely available in recent years. But the potential user of this abundance of riches faces two well-known major problems. The first is of an algorithmic nature: to what extent the computation can be executed in parallel and the raw power available used productively. The second is a matter of software tools, exacerbated by the facts that there are few standards for parallel software and that parallel computers have a short life: it takes much work to port and adapt software from one parallel machine to the next.

## 2. A programmer's workbench and library for parallel search

### 2.1. *On the nature of parallel search*

Most algorithms for solving search-intensive problems share common concepts and features often described in introductory textbooks in artificial intelligence. Among these, we recall:

- The problem is modeled as a state space: a problem graph where each node represents a configuration that potentially needs to be considered, and each edge, or “move”, transforms a configuration into a neighboring configuration. Typically, the size of the state space can be estimated a priori, but is so large that only small parts of it are represented explicitly at any one time, thus parts being visited must be recomputed on the fly. Typically, the structure of the space is irregular and unknown.
- The algorithm superposes a search tree on the edges of the graph and defines a traversal order, usually chosen among a few standard ones, such as depth-first, breadth-first, best-first, or iterative deepening.
- When any node is expanded (i.e., its neighbors are generated), each new neighbor defines a new search problem of the same type. Thus, search algorithms are easily parallelized. In addition, the fact that state spaces are typically enormous as compared to the number of processors gives the programmer complete control over the important parameter of granularity (i.e., the size of the subspace assigned to any one processor).
- New search tasks are generated everywhere, along the entire frontier of the graph visited so far, and can be processed almost independently of each other. This situation lends itself to distributed computation, thus avoiding the bottleneck typically associated with central control.

### 2.2. *Portable software for parallel computers*

Parallel computers differ greatly in their architecture in many respects, such as the number of processors, the topology of communication networks, memory manage-

ment, and many other factors that affect programming. Considering the rapidly changing nature of hardware development, a program optimized for one particular machine is likely to be obsolete within a few years. It is unreasonable to expect potential users of parallel computers, whose speciality is mostly the applications domain rather than systems development, to write programs optimized for particular machines.

Standardized interfaces that facilitate porting software among parallel machines of different architectures range from languages designed for parallel programming, such as High Performance FORTRAN (HPF) [17], to communication libraries, such as PVM [13] or MPI [15]. Most standards aim to be general-purpose platforms for any kind of programs and applications, but they strike different trade-offs between convenience to the programmer and achievable machine efficiency.

If one aims to provide simultaneously high convenience to the programmer and high efficiency, one has to drop some other desirable property – for example, that the interface be general purpose. It is evident that by narrowing the range of applications and/or algorithms to be run, one can exploit special properties of data structures and algorithms to achieve high efficiency.

This argument suggests that parallel computation may spread among general users more by the ready availability of convenient and powerful program libraries than by the development of parallel programming languages. In contrast to general-purpose languages, a program library is specialized towards a well-defined class of problems and algorithms. This narrow focus permits developers to optimize algorithms, once and for all, for parallel computers of a variety of common architectures.

Many experiments in parallel branch-and-bound have been made (see [8,14] for an overview). Most of them focus on particular applications or on various load-balancing mechanisms. The need for general-purpose parallel search algorithm libraries, though, was recognized as early as 1987 [10], and several libraries have been implemented [4,19,22,29]. Most of them are specialized to branch-and-bound. With ZRAM [20], we aim at a broader application range and in particular, at long computations. As far as we know, ZRAM is the first parallel search library to include reverse search and checkpointing.

### 2.3. Structure of ZRAM

ZRAM, a portable parallel library of exhaustive search algorithms, is designed as a case study to prove the feasibility of the goals listed above: portability, efficiency, and convenience of use. The goal of portability is achieved through ZRAM's architecture based on three interfaces that separate four layers of software (figure 1):

- **Message passing interface**

This interface hides all machine dependencies of the host systems. Since it has been modeled on a subset of the standard message passing interface MPI, we call it *MPI*<sup>-</sup>. It retains the basic nonblocking point-to-point send and receive primitives of MPI, but omits collective communication, communicators, etc.

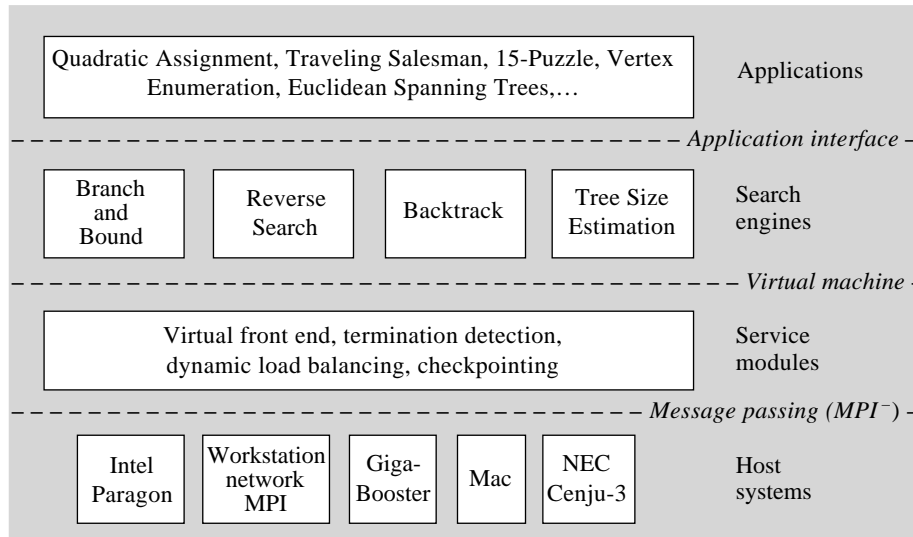


Figure 1. ZRAM architecture.

MPI<sup>-</sup> has been implemented for parallel computers as diverse as the Intel Paragon, NEC Cenju-3, ETH GigaBooster, and workstation networks using MPI. A single workstation or a Macintosh is treated as a parallel computer which has only one processor.

- **Virtual machine**

A layer of service modules enhances the message passing interface with functions common to many parallel programs, in particular to the search engines: dynamic load balancing, checkpointing, termination detection and a (virtual) front-end processor (section 3).

- **Application interface**

A library of Search Engines that run on top of the virtual machine makes up the layer that implements the top-most interface. Search algorithms (and the data structures they use) implemented so far include backtrack, branch-and-bound, reverse search [3], and a tree-size estimator based on sampling [16].

- **Applications**

Most ZRAM users will write programs only at this level: typically, some application-specific functions, such as branching rules, and a short, sequential main program that calls a search engine. Notice that there is no explicit parallelism in this layer! Application programs can range from simple to fairly elaborate, such as the two examples we will describe: a convex hull computation using reverse search and the branch-and-bound search for the hardest position in Sam Loyd's "15-puzzle" (section 5).

ZRAM is still under development. Interfaces to other machines and additional search algorithms will be implemented. The initial experience with ZRAM, however, already proves its capability to solve previously unsolved large-scale search problems efficiently, in parallel, with a modest investment in programming time.

### 3. The virtual machine: interface and common services

Defining a virtual machine is a time-honored technique to achieve portability by hiding most of the machine dependencies. A useful virtual machine must strike a balance between two competing goals:

- **High-level abstraction**  
A sufficiently high-level interface hides differences among various host systems and facilitates program development.
- **Efficiency**  
It must be possible to map the interface onto the target hosts with little loss of efficiency.

#### 3.1. Services provided

ZRAM is designed for a message passing model of parallel computation. The abstraction level of the two most popular message passing virtual machines, PVM and MPI, is too low for our purposes. The ZRAM virtual machine provides the following additional functions:

- (1) **Dynamic load balancing**  
In practice, the size and shape of a search tree is unknown at the beginning of a computation. Thus, it cannot be partitioned into subtrees of equal size a priori. Therefore, a parallel search algorithm needs a mechanism to redistribute the work among the processors during the computation when the sizes of subtrees become available and processors run out of work. As this issue is common to most tree search algorithms, dynamic load balancing is implemented in the virtual machine. At present, ZRAM offers two distinct load-balancing disciplines, a general one (relaxed queue, see section 3.2) and one specialized to best-first search (speculative priority queue).
- (2) **Checkpointing**  
Our computations take days or weeks on a powerful parallel computer. Since it is rarely possible to get that much processor time in one chunk, a checkpointing facility is necessary (see section 3.3). The combination of load balancing and checkpointing supports a dynamically changing number of processors: computations can be interrupted at a checkpoint and continued later with a different number of processors.

**(3) Distributed termination detection**

Many distributed algorithms need some way of detecting the condition when all subtasks spawned have terminated. ZRAM contains a standard algorithm [21] for global termination detection.

**(4) A (virtual) front-end processor**

Parallel programs often use a master process for managing sequential tasks, such as initialization, reading input and making an initial distribution of the work to other (slave) processors. Slaves execute a main loop consisting of a receive operation, a computation, and sending results back to the master. Hiding this main loop inside the virtual machine makes programs more readable. Hence, ZRAM defines a front-end process that executes the main program, and slave processes that execute their receive–compute–send loop. Every call to a search engine on the front end corresponds to one iteration of the loop by the slaves. In contrast to a pure master–slave model, the processing elements (slaves) may communicate with each other during the compute phase by using MPI’s point-to-point communication. Some host systems include a designated front-end processor, others do not. In the latter case, the front-end process and one compute process are assigned to the same physical processor.

**(5) Implicit receive operation**

Typical message passing programs contain receive operations followed by a switch statement on some message tag. This programming style is analogous to a (non-object-oriented) procedure that executes a switch on the type of its arguments, and has the same disadvantage (collecting information which does not belong together, lack of extensibility). Our virtual machine installs every message tag together with a *message handler* procedure, and the handler is called implicitly when a message is received.

**3.2. Dynamic load balancing implementation**

ZRAM’s dynamic load-balancing mechanism (relaxed queue) is currently used by all search engines except best-first branch-and-bound. Since centralized load balancing among many processors easily becomes a bottleneck, ZRAM balances computational loads in a distributed manner. Although we have implemented only one load-balancing algorithm, the algorithm could be replaced without changing the interface.

From an abstract point of view, the virtual machine manages just one (distributed) data structure: a global queue of *work units* (for depth-first branch-and-bound, a work unit is a subtree represented by its root node; for reverse search, it is an interval of the depth-first traversal of the tree). Every processor repeatedly removes a work unit from the global queue, works on it, and inserts zero or more new (smaller) work units into the queue. The algorithm terminates when the virtual machine detects that the global queue is empty.

Viewed locally, every processor manages its own local queue of work units. It can remove items from the queue and insert others. The coarse-grain parallelism prevalent in search algorithms allows simple load-balancing heuristics to be effective. Because a single node of the search tree almost always generates many new nodes, all processors are usually busy working off the initial node assigned to them. Only towards the very end of the computation do some processors become idle. When the local queue of a processor becomes empty, it sends an “I need work” message to some other randomly selected processor. If this second processor’s local queue contains at least two elements, it sends one of these back to the requesting processor. Otherwise, the second processor forwards the “I need work” message to a third processor. To keep the algorithm simple, the third and all succeeding processors are selected in a round-robin fashion rather than randomly.

### 3.3. Checkpointing implementation

Where is the best place to implement checkpointing? The same mechanism that transfers data between processors in order to balance the load is also used to save data to disk. At regular intervals (e.g., every hour) during the tree search, the computation is interrupted, the dynamic load-balancing algorithm is brought into a known state by synchronizing all processors, and the global queue of work units is saved to disk. The virtual machine then calls a function in the search engine to save other relevant data, such as current bounds. On machines supporting a signal facility, checkpointing can also be triggered by sending a signal to the process group before killing the job.

To restart the computation, the virtual machine first reads the global data and broadcasts it to all processors. It then reads the queue of work units and redistributes it onto the new set of processors. Finally, it calls a search engine function to read the other saved data.

## 4. Library of parallel search algorithms

### 4.1. Application interface

ZRAM search engines are implementations of general-purpose search algorithms, such as backtrack, branch-and-bound, and reverse search. Since branch-and-bound is a general paradigm and admits many technical variations, it is implemented by three distinct search engines.

There is no explicit parallelism at the application interface level. All the search engines share the same style of interface. When using a search engine, one has to

- define the global data to be distributed among the processors;
- define a data type for the nodes of the search tree;
- define the problem-specific routines that are to be called by the search engine (figure 2).



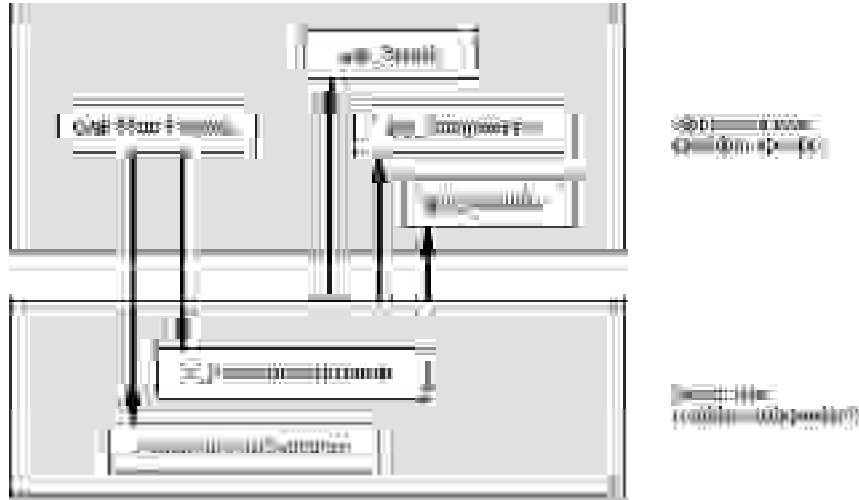


Figure 2. Call graph for the branch-and-bound interface.

The global data, which describes the problem instance, is initialized by the main program and supplied to the search engine as a parameter. It is then broadcast to all processors. It cannot be modified during the execution of the search algorithm.

ZRAM's search engines automatically distribute the nodes of the search tree among the processors. In contrast, the work to be done in a single node, which is application-dependent, is not parallelized. This approach is efficient for search trees much larger than the number of processors, whose coarse granularity does not limit achievable speedup [5]. The tree-size estimator calls the same problem-specific routines as the search engines.

Below, we describe the implementation of each search algorithm.

#### 4.2. Branch-and-bound

Branch-and-bound is a standard tool of combinatorial optimization, used to minimize or maximize a given objective function over some state space. Without loss of generality, we describe the minimization case. A branching rule is used to recursively divide a space into subspaces, thus generating a search tree. A relaxed version of the original problem, solved in each subspace, yields a lower bound for the optimal solution. Whenever the lower bound exceeds the currently known best solution, or an optimal solution of the subproblem is found, a cut-off occurs. Among various traversal orders of the search tree, best-first has the advantage that the resulting search tree contains the minimum number of nodes.

Every branch-and-bound search engine needs three application-dependent functions: a branching function, which also computes lower bounds of subnodes; a solution test; and a function which compares the lower bounds stored in two nodes.

We have implemented three variants of sequential and parallel branch-and-bound engines: *depth-first*, *iterative deepening*, and *best-first*. Depth-first and iterative deepening, which require a stack, balance computational load by using the load-balancing module described in section 3.2. Best-first, which requires a priority queue, is approximated by using a heuristic, speculative priority queue. In contrast to a priority queue, which returns a global minimum, a speculative queue returns a node close to minimum. This is a trade-off between the communication overhead involved in finding a global minimum and the search overhead caused by the expansion of non-minimum nodes.

The following applications have been implemented using the branch-and-bound engine:

- Traveling Salesman Problem using the Held–Karp 1-Tree as the lower bound, with iteration of Lagrangian coefficients [5].
- Quadratic Assignment Problem using the Gilmore–Lawler bound.
- Vertex cover.
- 15-puzzle.

We present the last application in detail in section 5.2.

#### 4.3. Reverse search

Reverse search [2,3] is a memory-efficient technique for enumerating the vertices of a graph without marking visited vertices. Its time complexity is linear in the output size and its space complexity is independent of the output size.

Suppose we have a finite connected graph  $G$  and an objective function to be maximized over its vertices. A *local search algorithm* is a procedure for moving from a vertex to an adjacent vertex whose objective function value is larger. Obviously, the local search algorithm finds a local optimum. The union of all paths that can be chosen by the local search algorithm is a partition of the graph into trees rooted at the local optima.

Reverse search reverses this process. Suppose that a graph  $G$  is given in the form of an *adjacency oracle* that returns all the neighbors of any given vertex, and suppose that we know how to enumerate all the local optima. Starting at each local optimum, we can traverse every tree in a depth-first manner, and thus enumerate all vertices of the graph.

The reverse search engine needs four application-dependent functions: the local search function, the adjacency oracle, an equality test for vertices, and the maximum degree in the graph.

Reverse search traverses a tree in depth-first order. It uses the adjacency oracle to move down the tree (away from the root) and the local search function to move up. In contrast to backtracking, which has a space complexity proportional to the height

of the tree, reverse search does not have to save the whole path from the current node back to the root, and thus its space complexity is independent of the height of the tree. ZRAM keeps a cache of ancestors of the current node which eliminates most calls to the local search function. Load balancing transfers subproblems (intervals) between processors, but not the ancestor cache. This loss of context contributes to the parallelization overhead.

The memory requirements of reverse search are independent of the size of the graph, and any intermediate state of the sequential computation can be saved by merely recording the current node, along with the values of global variables, such as bounds. Thus, checkpointing and restarting a computation are inexpensive operations, a great practical asset for long computations.

The large variance in the size of the subtrees, coupled with the impossibility of estimating their sizes, creates the need to balance the processor load dynamically. The reverse search engine uses the load balancer of ZRAM's virtual machine. The work is not split into contiguous subtrees, but rather into intervals of the tree's pre-order traversal. In this memory-efficient approach, every processor stores only one interval instead of a set of subtrees. Such an interval  $[s, e]$  extends from its start node  $s$  to its end node  $e$ . Two operations are defined on intervals: (1) removing the start node of an interval, and (2) splitting an interval of size greater than one into two parts  $[s, m]$  and  $[m + 1, e]$ .

We can limit our implementation to a set of intervals which contains the interval corresponding to the whole search tree and is closed under these two operations, so we work only with intervals representable by a triple  $\langle s, d, k \rangle$ , where  $d$  is the depth of the start node relative to the least common ancestor and  $k$  is the number of the node following  $e$  as a neighbor of the least common ancestor defined by the adjacency

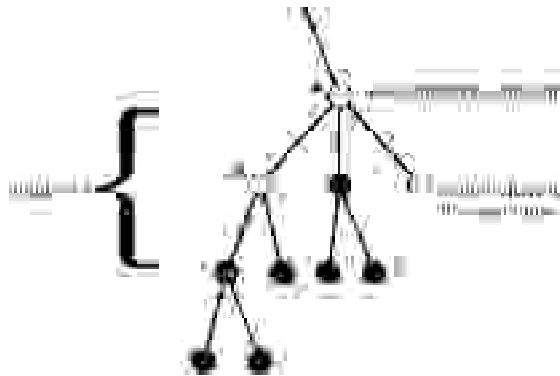


Figure 3. The filled circles mark an interval of the pre-order traversal represented by  $\langle s, 2, 3 \rangle$ . The nodes which may be in the ancestor cache are marked by a  $\times$ .

oracle (figure 3). The whole search tree is represented by the triple  $\langle \text{root}, 0, \infty \rangle$ . As the average size of a node is typically much greater than two integers, this implementation outperforms one which simply stores the start and the end node.

The following applications have been implemented using the ZRAM reverse search engine:

- convex hull and vertex enumerations in polyhedra [2];
- enumeration of connected induced subgraphs;
- enumeration of Euclidean spanning trees.

For other applications of reverse search, see [3]. In section 5.1, we present some computational results of the vertex enumeration application.

#### 4.4. Backtracking

Backtracking is a form of exhaustive search used to find all solutions to a problem. In contrast to branch-and-bound, no upper bound is saved in order to prune subtrees. This simplifies the interface (no comparison between nodes) and the implementation (no broadcasting of upper bounds). In fact, the application has to provide only a branching procedure, and the search engine takes care of the rest.

The following applications have been implemented using the backtrack code:

- the  $n$ -queens problem;
- enumeration of all partitions of a set.

#### 4.5. Tree size estimation

If one knew the running time of a given problem instance before starting the computation, one could avoid starting computations which later turn out to exceed the time available. This would be preferable to killing a computation which has already used many resources.

Since we cannot predict the running times of a branch-and-bound algorithm by traditional complexity analysis (aside from very weak worst-case considerations), another approach is needed for estimating the resources such as running time involved in the actual solution of an instance. The difficulty of a given instance is estimated in order to decide whether or not it is solvable by the available algorithm in a reasonable time.

Knuth's tree-size estimator [16] evaluates a relatively small number of paths of the search tree and computes the degree of every node along these paths. This information provides an unbiased estimate of the size of the full tree. In the ZRAM implementation, every processor independently follows some number of paths in the search tree and collects the data needed. At the end of the computation, one processor gathers the results and computes the mean and standard deviation.

The different paths in a search tree generally have different lengths and their evaluations have different running times. As in any other search engine, the total work of the estimator is balanced dynamically among the processors.

## 5. Applications and benchmarks

ZRAM, running on a variety of parallel computers, has been instrumental in solving search-intensive problems that are too large for sequential computers. We describe two of them:

- Convex hull and vertex enumeration in polyhedra (reverse search).
- The 15-puzzle (branch-and-bound).

### 5.1. Convex hull and vertex enumeration in polyhedra

A *convex polyhedron*, or simply *polyhedron*, is the solution set of a system of linear inequalities in  $d$  variables; it is a subset  $P$  of the  $d$ -dimensional space  $R^d$  of the form  $\{x \in R^d : Ax \leq b\}$ , for some matrix  $A \in R^{m \times d}$  and vector  $b \in R^m$ . The *vertex enumeration problem* asks for the generation of all vertices (extreme points) of  $P$  for given inputs  $A$  and  $b$ . The *convex hull problem* is the reverse problem, that is, for a given set  $V$  of  $m$  points in  $R^d$  find minimal  $A$  and  $b$  whose solution is the convex hull of  $V$ . These two problems are computationally equivalent (see [2,11]), and have been extensively studied in operations research and computational geometry. Many problems, such as the computation of the  $d$ -dimensional Voronoi diagram or the Delaunay triangulation, can be reduced to one of these problems, see [9].

The reverse search algorithm [2] has the best time and space complexities for solving these two problems, at least under the assumption of nondegeneracy. The time and space complexities of the vertex enumeration problem for  $n$  vertices are  $O(md \log n)$  and  $O(md)$ , respectively. Reverse search is ideally suited for parallel computation, unlike other known algorithms such as the double description method and its dual, the beneath-and-beyond method [9], which is a memory-intensive sequential algorithm.

Table 1 shows the running times for a sample polytope on four different machines. The polytope c7-3 is the cross product of the dual of the 7-dimensional hypercube and a 3-dimensional cube. It is 10-dimensional, has 134 facets and 112 vertices, and thus  $d = 10$ ,  $m = 134$  and  $n = 112$ . The reverse search algorithm generates 75040 bases in a search tree of depth 37. Note that each basis of the system  $Ax \leq b$  represents a vertex, but generally there are many bases representing the same vertex. Typical instances that arise in applications and have been solved using ZRAM are three orders of magnitude larger.

Figure 4 shows the speedup of parallel vertex enumeration on the Paragon MP (Cenju-3 results are similar). The speedup of 35.2 for 100 processors is rather good, considering the small test problem. Higher dimensional problems with much wider and larger search trees have a more favorable speedup. See section 4.3 for an explanation of the parallelization overhead.

Our parallel code was able to solve three large polytopes on a Cenju-3 with 64 processors (table 2). These instances could not be solved on any single workstation – estimated CPU time on a DEC AXP workstation ranges from 130 days to 5.4 years.

Table 1  
Running times for c7-3 on different machines.

Machine	Processors	Time [s]	Speedup
DEC AXP 3000/700	1	791.5	1.0
Paragon MP	1	2523.0	1.0
	10	268.3	9.4
	100	71.6	35.2
	150	65.7	38.4
Cenju-3	1	952.0	1.0
	10	101.5	9.4
	100	27.8	34.2
GigaBooster	1	1520.3	1.0
	7	248.3	6.1

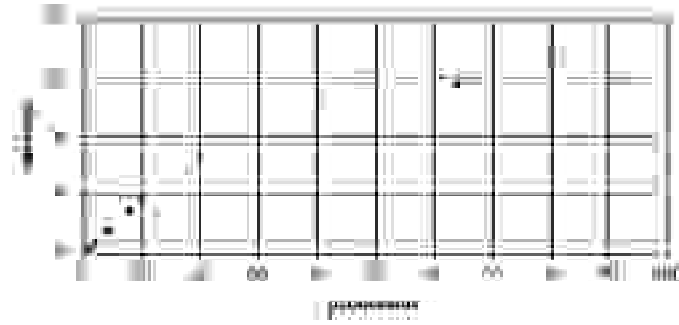


Figure 4. Speedup for c7-3 on the Intel Paragon.

Table 2  
Three previously unsolved polytopes.

Name	$d$	$m$		Bases	Time (64 processors)	Time on workstation (estimated)
01torus15x	14	240	101445	409794857	3 days	130 days
mit71-61	60	71	3149579	57613364	4.5 days	130 days
01torus16x	15	340	519275	3971059018	38 days	5.4 years

The Cenju-3 system used is a research machine dedicated primarily to software development. During the computation, it was rebooted with different system software at unexpected times. Thus, the restart capability of ZRAM running reverse search was essential.

## 5.2. The 15-puzzle

Games and puzzles often serve as ideal test cases for calibrating the effectiveness of “brute force” approaches to enumeration problems. Simple rules and problem instances that cover a wide range of difficulty have made puzzles standard benchmarks for measuring progress and for assessing the performance of various search techniques. The steadily rising computing power of available hardware has made it possible to attack problems which seemed unsolvable a few years ago. Examples of successful applications of exhaustive search techniques are chess endgames (where all five-stone endgames without pawns and some with pawns have been analyzed [26–28]), or the game of Nine Men’s Morris, which has been proved to be a draw [12]. The largest state spaces which have been search exhaustively are about  $10^{10}$  states (e.g., Nine Men’s Morris or the chess endgame KRNNKNN). The 15-puzzle state space contains about  $10^{13}$  positions and is a target for a new milestone in exhaustive search.

Sam Loyd’s 15-puzzle consists of a permutation of 15 numbered tiles and an empty space over a  $4 \times 4$  board (figure 5). Any tile adjacent to the empty space can slide into the empty space in a single move. The goal is to find, for any given starting



Figure 5. The 15-puzzle.

permutation, a minimum sequence of moves that leads to the goal state. The generalized  $n \times n$  version of this problem is NP-hard in the size of the board [24]. A branch-and-bound algorithm that uses the Manhattan distances as a lower bound and executes an iterative-deepening search has become the standard approach [18]. It has been enhanced by refinements, such as hashing techniques to avoid redundant computation [25], or databases of tight lower bounds [12].

The hardest positions (those requiring the maximum number of moves in a minimum solution sequence) were unknown previous to this work. Gasser [12] found 9 positions, requiring 80 moves, and proved that there are no positions requiring more than 87 moves. We have now proved that the hardest 15-puzzle positions require, in fact, 80 moves. We have also discovered two previously unknown positions, requiring exactly 80 moves to be solved.

Our search for the hardest 15-puzzle position iterates, for various values of  $k$ , two steps:

- Step 1. Generate** a candidate set that contains all positions  $p$  which may require more than  $k$  moves (and probably contains some positions requiring less than  $k$  moves);
- Step 2.** For each candidate, **prove** either that it can be excluded from the candidate set, or that it requires more than  $k$  moves.

The generation of the candidate set is done by applying to all positions  $p$  in the 15-puzzle a heuristic that returns a path from  $p$  to the goal positions, with its length  $h(p)$ . Whenever  $h(p) > k$ ,  $p$  becomes a candidate. The requirements on  $h$  are

- (A) It is fast, so it can be evaluated for all  $10^{13}$  positions.
- (B) It is a good upper bound, so the candidate set is not bloated by too many simple positions.

We met these requirements by computing and storing databases for many partial solutions. In a first step, a subset of the tiles are moved to their goal position. A first database contains the optimal values for all permutations of this subset. In the second step, the remaining tiles are moved to their goal positions without moving the tiles fixed in the first step. Again, a database contains the optimal values for all permutations of the remaining tiles on the remaining board. Experiments with different settings of the first subset of tiles have led us to choose the L-shaped rim consisting of the right-hand column and the bottom row of the  $4 \times 4$  board. Thus, the second step reduces to finding the optimal solutions of the  $3 \times 3$  puzzle.

Different approaches can be used during the proving phase. The obvious approach of actually solving each candidate requires too much time. A shortcut for reducing the candidate set applies consistency constraints  $CC$ . As an example, let  $c$  be a candidate and  $n_i$  its adjacent positions (i.e., those that can be reached from  $c$  in a single move). If  $\min(h(n_i)) < h(c) - 1$ , a shorter path for  $c$  can be found.

Both steps of the algorithm are computationally intense. Using ZRAM on 64 nodes of an Intel Paragon, we have computed the candidate set shown in table 3.

Table 3

Number of candidates in the generated candidate set.

$k$	$h(c)$	Before applying $CC$	After applying $CC$
79	80	34189	33208
	81	3792	1339
	82	393	44
	83	8	0
	84	1	0
	85	0	0



The remaining 1383 candidates, which might require 81 or more moves, have been solved in parallel by the standard branch-and-bound algorithm. Additionally, we used a depth-14 move generator to avoid a single 15-puzzle position appearing several times in the search tree, thus reducing the number of nodes in the very large search trees by roughly a factor of four compared to the standard approach. These computations were carried out on the NEC Cenju-3 with from 64 up to 128 processors and the solution of the 1383 instances took approximately three days on Cenju-3, which is the equivalent of 230 days of (sequential) CPU time. Thus, the computation would have been impossible within reasonable time without the use of a parallel computer system. The computation has shown that all 1383 candidates require less than 81 moves to be solved. Six positions required exactly 80 moves, 4 of them have previously been detected by Gasser [12]. The new ones are (15, 14, 13, 12, 10, 11, 8, 9, 2, 6, 5, 1, 3, 7, 4, 0) and (15, 11, 13, 12, 14, 10, 9, 5, 2, 6, 8, 1, 3, 7, 4, 0). The computation proves that the hardest 15-puzzle positions require exactly 80 moves to be solved.

## 6. An emerging tool

Our portable library of parallel search algorithms ZRAM, although still under development, has already provided several useful functions. Thanks to the straightforward porting to half a dozen machines of different architectures, ZRAM has served as a rapid-prototyping tool for benchmark tests that compare the performance of various computers.

However, our aim in developing ZRAM is not merely to facilitate computational experiments, but rather to provide a tool that permits users who are not specialists in parallel computing to solve application problems they cannot solve by other means. Admittedly, for a limited class of problems, namely those that are dominated by search, ZRAM harnesses the extensive computing power of parallel machines in an efficient manner, without special efforts on the part of the applications programmer. As we have shown, the search algorithms built into ZRAM often achieve a linear speedup over a wide range of problem sizes and number of processors.

The goal of providing a useful tool has already been achieved. One of the vertex enumeration calculations described (mit71-61) arises from previously unsolved materials science problems presented by Garbulsky of MIT [7]. The solitaire cone calculations (01torus15x and 01torus16x) confirm conjectures by Deza (Ecole Polytechnique Fédérale de Lausanne, Switzerland).

Other programmers are now extending the library of search algorithms. In due time, we expect ZRAM to include a comprehensive library that contains most general-purpose search techniques. We encourage readers to experiment with it. The ZRAM source code is available at <http://www.jn.inf.ethz.ch/ambros/zram/zram.html>.

## Acknowledgements

We thank David Avis for permission to use and modify his “Irs” code, which we have integrated into ZRAM, Will Sawyer for his help with porting ZRAM to the NEC

Cenju, the Swiss Center for Scientific Computation for the extensive use of their facilities, Tony Gunzinger for making available a parallel computer GigaBooster, and the Swiss National Science Foundation for financial support.

## References

- [1] K. Appell and W. Haken, The solution of the four-color-map problem, *Sci. American* (Oct. 1977) 108–121.
- [2] D. Avis and K. Fukuda, A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra, *Discrete Computational Geometry* 8(1992)295–313.
- [3] D. Avis and K. Fukuda, Reverse search for enumeration, *Discrete Applied Mathematics* 65(1996) 21–46.
- [4] M. Benaïchouche, V. Cung, S. Dowaji, B. Le Cun, T. Mautor and C. Roucairol, Building a parallel branch and bound library, in: *Solving Combinatorial Optimization Problems in Parallel*, LNCS 1054, eds. A. Ferreira and P. Pardalos, Springer, Berlin, 1996, pp. 201–231.
- [5] A. Brügger, A parallel best-first branch and bound algorithm for the traveling salesperson problem, in: *Proceedings of the 9th International Parallel Processing Symposium, Workshop on Solving Irregular Problems on Distributed Memory Machines*, ed. S. Ranka, 1995, pp. 98–106.
- [6] A. Brügger, Solving hard combinatorial optimization problems in parallel: Two case studies, Ph.D. Thesis, ETH Zürich, 1997.
- [7] G. Ceder, G.D. Garbulsky, D. Avis and K. Fukuda, Ground states of a ternary fcc lattice model with nearest- and next-nearest-neighbor interactions, *Physical Review* B49(1994)1–7.
- [8] R. Corrêa and A. Ferreira, Parallel best-first branch-and-bound in discrete optimization: A framework, in: *Solving Combinatorial Optimization Problems in Parallel*, LNCS 1054, eds. A. Ferreira and P. Pardalos, Springer, Berlin, 1996, pp. 171–200.
- [9] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer, 1987.
- [10] R. Finkel and U. Manber, DIB – a distributed implementation of backtracking, *ACM Transactions on Programming Languages and Systems* 9(1987)235–256.
- [11] K. Fukuda and A. Prodon, Double description method revisited, to appear in: *Lecture Notes in Computer Science*, Springer. PS file available from ifor13.ethz.ch (129.132.154.13), directory/pub/fukuda/reports.
- [12] R. Gasser, Harnessing computational resources for efficient exhaustive search, Ph.D. Thesis, ETH Zürich, 1995.
- [13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manček and V. Sunderam, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.
- [14] B. Gendron and T.G. Crainic, Parallel branch-and-bound algorithms. Survey and synthesis, *Operations Research* 42(1994)1042–1066.
- [15] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.
- [16] D.E. Knuth, Estimating the efficiency of backtrack programs, *Math. Comp.* 29(1975)121–136.
- [17] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Stelle, Jr. and M.E. Zosel, *The High Performance FORTRAN Handbook*, MIT Press, 1994.
- [18] R.E. Korf, Depth-first iterative deepening: An optimal admissible tree search, *Artificial Intelligence* 62(1993)97–109.
- [19] N. Kuck, M. Middendorf and H. Schmeck, Generic branch-and-bound on a network of transputers, in: *Transputer Applications and Systems '93*, eds. R. Grebe et al., IOS Press, Amsterdam, 1993, pp. 521–535.
- [20] A. Marzetta, ZRAM: A library of parallel search algorithms and its use in enumeration and combinatorial optimization, Ph.D. Thesis, ETH Zürich, 1998. <http://www.inf.ethz.ch/publications/diss.html>.

- [21] F. Mattern, Experience with a new distributed termination detection algorithm, in: *Distributed Algorithms 1987*, LNCS 312, Springer, 1987, pp. 127–143.
- [22] G.P. McKeown, V.J. Rayward-Smith and H.J. Turpin, Branch-and-bound as a higher-order function, *Annals of Operations Research* 33(1991)379–402.
- [23] J. Nievergelt, R. Gasser, F. Mäser and C. Wirth, All the needles in a haystack: Can exhaustive search overcome combinatorial chaos?, Invited paper in *Lecture Notes in Computer Science 1000*, *Computer Science Today*, ed. J. van Leeuwen, Springer, 1995, pp. 254–274.
- [24] D. Ratner and M. Warmuth, Finding a shortest solution for the  $(N \times N)$ -extension of the 15-puzzle is intractable, *Journal of Symbolic Computation* 10(1990)111–137.
- [25] A. Reinefeld and T.A. Marsland, Enhanced iterative-deepening search, *Reihe Informatik 120*, Paderborn Center for Parallel Computing, 1993.
- [26] L.B. Stiller, Exploiting symmetry on parallel architectures, Ph.D. Thesis, The Johns Hopkins University, 1995.
- [27] K. Thompson, Chess endgames, vol. 1, *ICCA Journal* 14(1991)22.
- [28] K. Thompson, Chess endgames, vol. 2, *ICCA Journal* 15(1992)149.
- [29] S. Tschöke and T. Polzer, Portable parallel branch-and-bound library: User manual, Technical Report, University of Paderborn, 1995.