Diss. ETH No. 12699

ZRAM: A Library of Parallel Search Algorithms and Its Use in Enumeration and Combinatorial Optimization

Dissertation

submitted to the
Swiss Federal Institute of Technology
Zurich

for the degree of
Doctor of Technical Sciences

presented by
Ambros Marzetta
Dipl. Informatik-Ing. ETH
born on April 8th, 1968
citizen of Basel BS

Accepted on the recommendation of
Prof. Dr. Jürg Nievergelt, examiner
Prof. Dr. Komei Fukuda, coexaminer

1998

Contents

List of Figures

Abstract

ZRAM is a software library which renders the power of parallel computers usable for combinatorial optimization and enumeration. It provides a set of parallel search engines for branch-and-bound, reverse search, backtracking and tree-size estimation. These search engines hide the complexity of parallelism from the application programmer.

ZRAM contains the first parallel implementation of the reverse-search algorithm and the first parallel branch-and-bound engine that can be restarted at checkpoints. It is the first search library containing a tree-size estimation tool, which proved to be valuable in allocating the limited CPU resources to the most promising problem instances. The combination of these elements, together with the efficiency of the implementation, allowed us to solve large enumeration and combinatorial optimization problems. These benchmarks include quadratic assignment instances which were previously unsolved and the enumeration of the vertices of complex high-dimensional polytopes.

ZRAM has proved its flexibility during the development of a wide range of applications (e.g., quadratic assignment problem, vertex and facet enumeration, hyperplane arrangements, 15-puzzle, Euclidean spanning trees, connected induced subgraphs). Some of its users had little or no experience in parallel programming and got access to parallel computers only through ZRAM.

The work on ZRAM has clarified what properties we require of a parallel search library and demonstrates that a four-layered structure (applications, search engines, common services, host systems) is a suitable architecture.

Kurzfassung

ZRAM ist eine Programmbibliothek, welche die Leistung paralleler Rechner dem Gebiet der kombinatorischen Optimierungs- und Aufzählungsalgorithmen zugänglich macht. Es enthält eine Sammlung paralleler Suchmaschinen für Branch-and-Bound, Reverse-Search und Backtracking sowie ein Werkzeug zur Grössenabschätzung von Suchbäumen. Die Schnittstelle der Bibliothek verbirgt die Komplexität der Parallelisierung vor dem Programmierer.

ZRAM enthält die erste parallele Implementierung des Reverse-Search-Algorithmus und die erste parallele Branch-and-Bound-Maschine, die man unterbrechen und neu starten kann. Es enthält auch als erste Bibliothek paralleler Suchalgorithmen ein Werkzeug, das die Grösse von Suchbäumen abschätzt. Dieses Werkzeug dient dazu, die beschränkten Ressourcen an Rechenzeit den erfolgversprechendsten Berechnungen zuzuteilen. Die Kombination dieser Elemente und deren effiziente Implementierung ermöglichten es uns, grosse kombinatorische Optimierungs- und Aufzählungsprobleme zu lösen. Darunter sind bisher ungelöste Instanzen des quadratischen Zuordnungsproblems und die Aufzählung der Ecken hochdimensionaler Polyeder.

Die Flexibilität von ZRAM zeigt sich im breiten Spektrum der Anwendungen (u.a. quadratisches Zuordnungsproblem, Aufzählung von Ecken und Facetten eines Polyeders, Arrangements von Hyperebenen, 15-Puzzle, euklidische Spannbäume, zusammenhängende Teilgraphen). Einige seiner Anwender hatten wenig oder gar keine Erfahrung mit parallelem Programmieren und bekamen erst dank ZRAM Zugang zu Parallelrechnern.

Anhand von ZRAM haben wir gezeigt, was für Anforderungen an eine Bibliothek paralleler Suchalgorithmen zu stellen sind und dass eine Strukturierung in die vier Schichten Anwendung, Suchmaschinen, virtuelle Maschine und Systemanpassung sinnvoll ist.

Acknowledgments

I sincerely thank the many people who assisted me in creating ZRAM and those who taught me valuable skills.

Jürg Nievergelt, my advisor, taught me to present scientific work effectively, insisting that I deliver my message clearly, in written work as well as in presentations. He left me complete freedom in my research and provided me with enough computing power for my projects.

Komei Fukuda, my coexaminer, sparked my interest in polyhedra and in reverse search algorithms. His wide range of knowledge and willingness to share it led to interesting research problems.

Adrian Brüngger was the first user of ZRAM, implementing the 15-puzzle, QAP, and TSP applications. The first user is always the one who suffers the most from bugs and deficiencies in the code. We had many technical discussions which greatly influenced the design of ZRAM.

Nora Sleumer used ZRAM to compute hyperplane arrangements, suggested various improvements to the library, and, having read the first fragments of my manuscript, showed me how to improve my writing English.

Three further users of ZRAM who gave me important feedback and motivation are *Frank Ammeter, Gerard Kindervater* and *Danuta Sosnowska.*

David Avis and *Jens Clausen* gave permission to use, modify and parallelize their programs for the vertex enumeration and quadratic assignment problems, respectively, and in this way laid the ground work for the two most prominent applications of ZRAM.

I have always enjoyed working in the pleasant environment of our research group: *Michele De Lorenzi* taught me to distinguish good movies from bad ones. *Ralph Gasser* showed me how to apply search algorithms to games with incomplete information. *Thomas Lincke* often questioned my technical assertions, forcing me to rethink them. *Fabian Mäser* regularly reminded me that combinatorial game theory is an interesting topic. *Martin Müller* gave me the helpful advice of an experienced researcher in many situations. *Matthias Müller,* the expert for computability theory, had enough patience to help me in preparing for important Jass matches. *Peter Schorn* persuaded me to join the Studienkommission, where I learnt a lot about bureaucracy. *Christoph Wirth,* the Macintosh expert, initiated our commercial project Stöck Wyys Stich.

Finally, I thank my family and all my other friends for their support.

Chapter 1

Project Overview

1.1 Goals

Search algorithms are a ubiquitous topic in computer science. They are the only method for the solution of most combinatorial optimization problems, serve for the enumeration of geometric objects, and are applied in artificial intelligence and game theory. They are an ideal base of a parallel software library because they are prevalent in diverse fields of computer science, because they need huge amounts of computing power, and because application-independent paradigms can be identified. These facts led us to define the following three goals for the ZRAM project:

- Developing a library of parallel search algorithms which can be used easily and successfully even by people who have no parallel programming experience.

- Ensuring portability of the library so that its lifetime exceeds that of parallel computer hardware.

- Demonstrating (by solving benchmark problems) that programs using our library are reasonably efficient compared to direct special-purpose implementations of similar algorithms.

1.2 Principal Findings

With ZRAM, we have implemented a software package containing among a variety of search algorithms the first parallel implementation of the reverse-search paradigm. During the development of the library and the experiments with various applications we could assert and substantiate the following theses:

- Automatic parallelization—that is, efficiently running a program written for a sequential computer on a parallel machine—is practicable in restricted cases, namely for combinatorial search algorithms. People who want to solve combinatorial problems without having learnt parallel programming are best served by a library which hides all parallel algorithms, rather than a general-purpose parallel language (which is difficult to learn) or a parallelizing compiler (which does not handle combinatorial search algorithms effectively).

- Large, search-intensive computations are well suited to parallel execution and produce good speedup. As the computation–communication ratio is high, neither a shared memory nor a fine-tuned load-balancing algorithm are necessary.

- In typical search applications the overhead created by the use of a library is negligible. Two ZRAM applications solved problem instances that had never been solved before: We enumerated the vertices of three large polytopes and solved nine previously unsolved benchmark instances of the quadratic assignment problem.

In particular, the following questions are of interest, and have been answered:

- How can we structure a library of parallel search algorithms to enhance clarity and usability of the code? What layers are necessary and what functions belong to which layer? How do interfaces to general parallel tree search engines look?

- What components, beside the search engines and the load-balancing code, must a library of search algorithms have? The quadratic assignment program has shown that a tree-size estimation tool and the possibility of restarting partial computations (checkpointing) are necessary tools for the solution of large problem instances.

- How much time does an inexperienced user need to learn the ZRAM interfaces and to develop a parallel program on top of it? Observation of the first few users of the library has shown that between one and three days are enough to produce a first parallel implementation on top of ZRAM. In comparison, an implementation of the load-balancing code from scratch would take months.

1.3 Structure of This Thesis

After an example of a complete ZRAM application (the n-queens program in Chapter 2), we give the necessary background on search algorithms as well as on models, tools and libraries in parallel computing (Chapter 3). Continuing, we explain the design and architecture of our library (Chapter 4). Then we describe the three main layers of ZRAM bottom-up: the virtual machine and common services in Chapter 5, the search-engine layer in Chapter 6, and finally the applications in Chapter 7.

Chapter 2

The N-Queens Example on ZRAM

The best way of introducing the programmer's interface to a software library is giving a small application as an example. For its simplicity, we have chosen a backtracking algorithm for the n-queens problem: Place n queens on a chessboard of size $n \times n$ such that every row, column and diagonal contains at most one queen. Our program has to enumerate *all* solutions for a given n. It does not eliminate symmetric solutions, as such a symmetry check would not help us in describing the interface, but decrease the legibility of the code. The program is written in ANSI C. Figure 2.1 shows the 40 solutions of the 7-queens problem.

Wanting to avoid variable-sized data types for this introduction, we define the maximum size of the board as `MaxN`:

```
#define MaxN 35
```

2.1 Data Types

Every ZRAM application (backtrack, branch and bound or reverse search) defines two data types, namely the global data and the node of the search tree. ZRAM expects both data types to be structs with a fixed header (`Z_NODE_HEADER`). The header contains information which is necessary for copying the data and for its encapsulation into messages sent among processors. It is initialized by the function `z_NewNode`, which is used instead of `malloc` for allocating variables of these types.

The global data consists only of the number of queens to place (the integer n). It is initialized once before the search engine is started and not modified during the search:

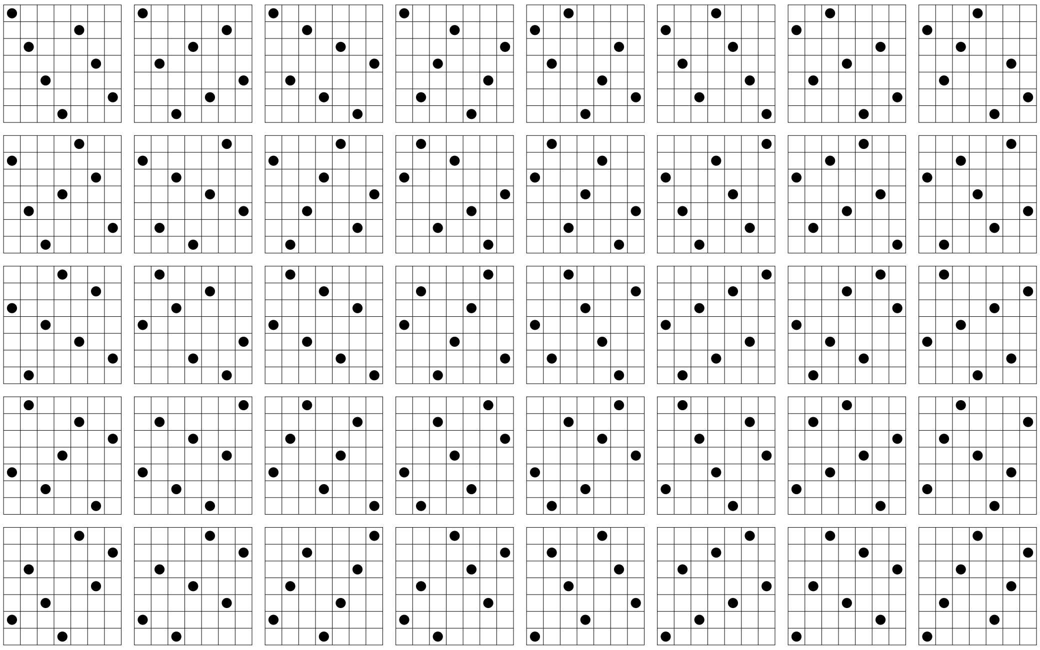

Figure 2.1: The 40 solutions of the 7-queens problem. The small C program in this chapter computes them in parallel.

```
typedef struct nq_Global { /* global data */
  Z_NODE_HEADER;
  int n;                    /* must be > 0 */
} nq_Global, *nq_GlobalP;
```

As the backtracking algorithm proceeds by placing queens into successive columns from left to right, a search-tree node is defined by d, the number of queens already placed (in the range 0 through n) and an array p of their positions. The sets of the occupied rows and diagonals are redundant and stored for efficiency:

```
typedef struct nq_Node { /* node of the search tree */
  Z_NODE_HEADER;
  int d;                /* 0 <= d <= n */
  int p[MaxN];            /* p[0..d-1] are in use */
  int row_occupied[MaxN],
      diag1_occupied[2*MaxN],
      diag2_occupied[2*MaxN];
} nq_Node, *nq_NodeP;
```

2.2 Procedures

We begin by showing the procedure `nq_Print`, which can output any node of the search tree, but is called by `nq_BranchInPlace` only for boards which have all n queens placed (i.e., nodes with $d = n$).

```
static void nq_Print(nq_GlobalP global, nq_NodeP node)
{
  int i;

  for (i=0;i<node->d;i++) printf("%3ld", (long)node->p[i]);
  printf("\n");
} /* nq_Print */
```

Now we are ready to understand the core of the program, the procedure `nq_BranchInPlace`. This procedure is called by ZRAM's parallel search engine (Figure 2.2). It is the procedure which increments d, tries to place the next queen in all positions of column d, and calls itself recursively. Only here, in the recursive call, the ZRAM version differs from the ordinary program shown in textbooks. Instead of calling itself directly, the procedure calls the library procedure `whatToDo` and gives the control to the parallel search engine. `whatToDo` does any one of calling `nq_BranchInPlace` recursively, sending the node to another processor immediately, storing the node for later evaluation and load balancing, or saving it in a disk file (checkpointing). The user does not see which of these possibilities is chosen by the search engine.

```
boolean nq_BranchInPlace(nq_GlobalP global, nq_NodeP node,
                         z_bt_DoProc whatToDo, void *ref)
{
  int col, row, *p;

  col = node->d;
  p = &node->p[col];
  node->d++;
  for (row = global->n-1; row>=0; --row)
    if (! node->row_occupied[row] &&
        ! node->diag1_occupied[MaxN + col + row] &&
        ! node->diag2_occupied[MaxN - col + row]) {
      *p = row;
      if (node->d == global->n)
        nq_Print(global, node);
```

```
      else {
        node->row_occupied[row] = 1;
        node->diag1_occupied[MaxN + col + row]++;
        node->diag2_occupied[MaxN - col + row]++;
        whatToDo((z_NodeP) node, ref);
        node->row_occupied[row] = 0;
        node->diag1_occupied[MaxN + col + row]--;
        node->diag2_occupied[MaxN - col + row]--;
      }
    }
  node->d--;
  return 0;
} /* nq_BranchInPlace */
```

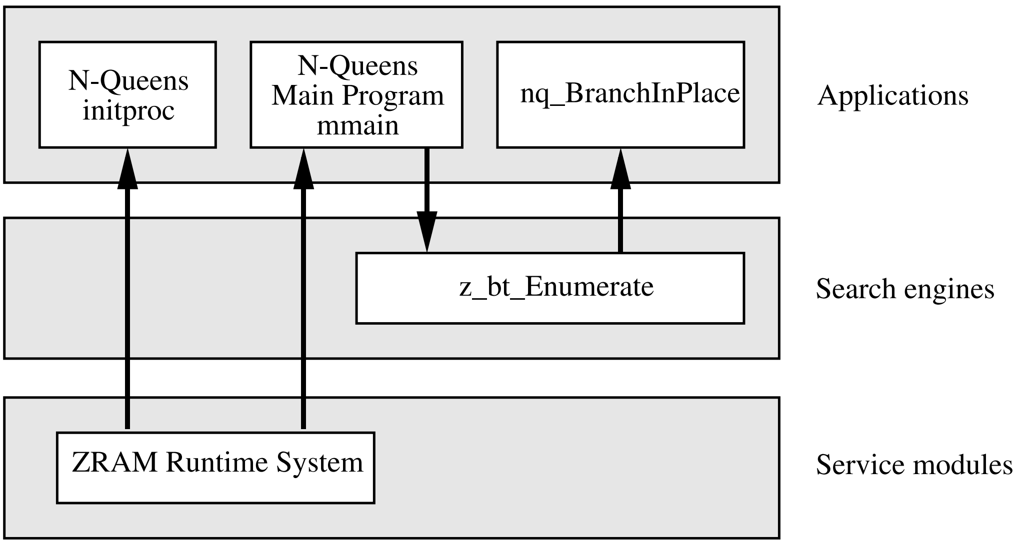

Figure 2.2: Call graph of the n-queens application with respect to ZRAM's layered structure. In the current chapter, we show the code of the application layer and how it interfaces with the lower layers.

A ZRAM backtracking application is completely specified by the procedure `BranchInPlace`. A reverse-search application specifies its search tree by means of an adjacency oracle and a local search function, whereas a branch-and-bound application defines a branching procedure, which divides a problem into subproblems, and a comparison function, which compares the lower bounds of two tree nodes.

2.3 Initialization

Every ZRAM application defines a procedure `initproc` which is executed by every processor of the parallel computer before the main program is started (see Figure 2.2). This procedure has four tasks:

1. Initialize the ZRAM modules which are used by the program. The n-queens program initializes the backtracking module by calling `z_bt2_-Init`.
2. Define the application-specific data types (`nq_Global` and `nq_Node`) to the system by calling `z_InstallClass`. The result of this operation is stored in `nq_GlobalTag` and `nq_NodeTag` and later used for data allocation by `z_NewNode`.
3. Define the application-specific procedures (`nq_BranchInPlace`) to the system by storing their address in a struct of type `z_bt_ProcSetT` and calling `z_InstallProcSet`. The result of this operation is later used in the call to the parallel search engine.
4. Do application-specific initializations. The n-queens program contains none of them.

```
static int nq_GlobalTag, nq_NodeTag;

static z_bt_ProcSetT nq_procset;
static int nq_procsetnum;

void initproc(void)
{
  /* This function is called once on every processor. */

  z_bt2_Init(); /* Initialize the search engine module */

  nq_GlobalTag = z_InstallClass(nq_Global, 0, 0, 0);
  nq_NodeTag = z_InstallClass(nq_Node, 0, 0, 0);
  nq_procset.bt_BranchInPlace = (z_bt_BranchProc)nq_BranchInPlace;
  nq_procsetnum = z_InstallProcSet((z_ProcSetP) &nq_procset);
} /* initproc */
```

Other search libraries simplify the initialization, but lose flexibility, by requiring fixed names for the data types and procedures. When their approach is used, the information given by `initproc` is found by the linker automatically. However, they lose the possibility of using more than one search engine with different sets of application-specific procedures in the same program (see Sec. 4.2.3).

2.4 Main Program

The main program of a ZRAM application has the name `mmain` (the identifier `main` is already used by ZRAM itself and cannot be used again). ZRAM starts the application such that `mmain` is executed by one processor after `initproc` has been executed by all processors. The main program allocates and initializes the global data and the root of the search tree (an empty chessboard), and calls the parallel backtrack engine `z_bt_EnumerateDFExp` (see Figure 2.2).

```
int mmain(int argc, char **argv, char **envp)
{
  nq_GlobalP global;
  nq_NodeP root;
  int i;

  global = (nq_GlobalP) z_NewNode(nq_GlobalTag,
                                   sizeof(nq_Global));
  if (!global) exit(1);
  global->n = atoi(argv[1]);
  if (global->n < 1 || global->n > MaxN) return Usage(argv[0]);

  root = (nq_NodeP) z_NewNode(nq_NodeTag, sizeof(nq_Node));
  root->d = 0;
  for (i = 0; i < global->n; i++) root->row_occupied[i] = 0;
  for (i = 0; i < 2*MaxN; i++)
    root->diag1_occupied[i] = root->diag2_occupied[i] = 0;

  z_bt_EnumerateDFExp(nq_procsetnum, (z_bt_GlobalP) global,
                      (z_NodeP)root, 0);

  return 0;
} /* mmain */
```

That's enough for the introduction. The presented code, linked together with ZRAM's backtrack engine, suffices to enumerate the n-queens configurations in parallel. In Section 6.1.1 we will show how a backtrack computation can be checkpointed and restarted.

Chapter 3

Background: Search and Parallel Computing

3.1 Search Algorithms

Computer science and operations research use the word *search* with various meanings. Four of them are:

- Searching a document in a collection of documents (information retrieval). This kind of search has become very common through the growth of the World Wide Web.
- Searching a record with a given key in a data base, such as a name in a phone directory or a word in a German–English dictionary. The importance of this problem has led Knuth to devote a full chapter of *The Art of Computer Programming* [40] to it.
- Searching a target in a continuous or discrete space [62]. Given are a probability distribution for the target's position in the space and a detection function that relates effort applied in a region to the probability of detecting the target given that it is in that region. The problem is to find an allocation of effort to space that maximizes the probability of detecting the target subject to a given constraint on effort. A typical application is searching a lost submarine in the ocean.
- Searching elements with given properties in graphs. In general, these graphs are not stored explicitly in memory, but implicitly given by a rule for computing neighbors of a point. Often, an objective function to be minimized or maximized is defined on the points of the graph (as in combinatorial optimization or in two-player games), or all elements

have to be enumerated (as in computing the vertices of a polyhedron given by its facets). This kind of search is investigated in the present dissertation. It has a huge number of applications ranging from the more than 300 NP-complete problems listed by Garey and Johnson [27] to strategy games (e.g., Chess or Go) and enumeration problems in geometry.

Sometimes, we call the graph being searched a *state space* and its vertices *states* or *nodes.* In combinatorial optimization, where a problem is decomposed into subproblems, the states correspond to the subproblems.

3.1.1 Classification

Graph search algorithms can be classified by the characteristic of *search completeness* [28]. It makes a big difference whether an approximate solution is satisfactory or we need a provably optimal one. Approximate (near-optimal) solutions are found by *heuristic search algorithms* (e.g., simulated annealing or GRASP), which search a random subset of the state space and thereby sacrifice accuracy for speed (Sec. 3.1.5). Exact (optimal) solutions are found by *exhaustive search algorithms* (e.g., backtracking) which either search the complete state space or cut off subspaces that provably do not contain solutions.

Exhaustive search algorithms are classified further by *search direction. Forward search* (Sec. 3.1.2) starts from states with unknown value and tries to find their values. In contrast, *backward search* (Sec. 3.1.3)—not to be confused with reverse search—starts from states whose value is known and propagates this information to unknown states. Combinations of forward and backward search are possible (Sec. 3.1.4).

3.1.2 Forward Search

Forward search is the most important class of exhaustive search techniques, and so far, all of ZRAM's search engines belong to it. It often finds exact results to NP-complete problems efficiently, although we cannot prove this to be true in general. When we look at the difficulty of parallelizing forward-search algorithms, we can group them—depending on the amount of global information used—into four classes of increasing difficulty level (Fig. 3.1).

1. The most basic form of forward search is the well-known backtracking, where a fixed search tree, of which some leaves are defined to be solutions, is traversed in depth-first order. If all solutions are to be

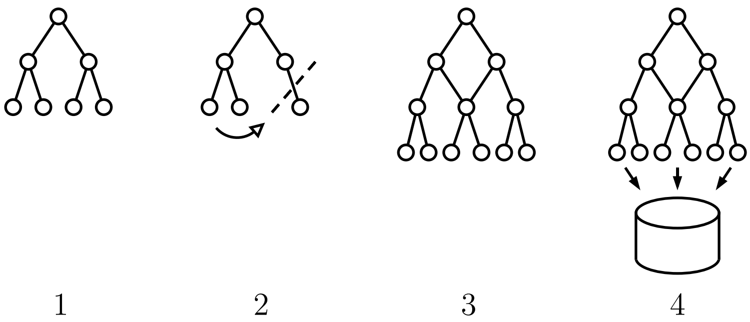

Figure 3.1: The four classes of forward search algorithms: 1. Independent subtrees. 2. Cutoffs. 3. Cycles. 4. More global information.

enumerated, the whole tree must be traversed and the ordering of the subtrees is irrelevant. If only one solution is to be found, a good subtree ordering heuristic can accelerate the algorithm.

Independence of subtrees is the property which identifies this class and facilitates parallelization. The reverse-search paradigm shares this property. In a parallel implementation of the other three classes, the dependencies among subtrees can lead to search overhead.

2. Our second class introduces a dependence among the subtrees, but still forbids cycles in the state space. In optimization and game trees, an objective function assigns every solution a value, and we do not only want to find any solution, but the one which has the lowest value. Both branch-and-bound and the alpha-beta algorithm cut off subtrees depending on the best solution currently known *(upper bound).* A good search order (move ordering heuristic) is important to find good solutions quickly and to cut off subtrees as large as possible. This fact complicates the parallelization (of alpha-beta even more than branch-and-bound) because the parallel evaluation reduces the total search order to a partial one and because the programmer has to ensure that all processors know the value of the (global) current best solution.

3. Whereas the first two classes of algorithms work on trees, many state spaces are graphs containing cycles. A search algorithm for such general graphs has (1) to ensure that it does not get into an endless loop and (2) to identify identical states for efficiency. To this end, it saves the value of some or all states in a *transposition table* or *hash table* (for an application to optimization, see [17]). A global hash table poses even greater challenges to the parallelization than a global upper bound, and there are few good results in this area [66].

4. Certain parallel search algorithms need even more global information than an upper bound and a hash table. For instance, Applegate's parallel branch-and-cut algorithm for the traveling salesman problem maintains a global cutting-plane cache. He reports that the algorithm has low communication overhead because it spends much time in computing linear programs and cut-finding heuristics for every subproblem. The branch-and-cut library ABACUS [64] might be a good starting point for designing a general parallel library of such algorithms.

An important issue in every parallel search algorithm is dynamic load balancing, as subtrees differ in size by orders of magnitude, a fact which makes it impossible to distribute the workload evenly when the computation starts.

Many experiments in parallel branch-and-bound have been made (see [18, 30] for an overview). Most of them concentrate 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 [25], and several libraries have been implemented (Table 3.1). Most of them are specialized to branch-and-bound. With ZRAM, 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. It includes search engines belonging to the first two classes of forward-search algorithms.

Name	Topic	Authors
—	Branch and bound	McKeown et al. (East Anglia) [46]
—	Branch and bound	Kuck et al. (Karlsruhe) [41]
BOB	Branch and bound	Cung et al. (Versailles) [5]
DIB	Backtracking	Finkel and Manber [25]
PPBB	Branch and bound	Tschöke and Polzer (Paderborn) [67]
PUBB	Branch and bound	Shinano (Tokyo) [57, 56]
PIGSeL	Depth-first search	Sanders (Karlsruhe) [55]
ZRAM	Tree-search algorithms	Marzetta (ETH Zürich)

Table 3.1: Parallel search libraries.

3.1.3 Backward Search

In certain search problems, it is preferable to begin the computation not in a state with unknown value, but rather in the goal states whose values are known. This technique, which is called *backward search* or *retrograde analysis,* simplifies the detection of identical subtrees which is often necessary in forward search algorithms. It trades space in exchange for a lower execution time.

For problems which exhibit a symmetry between start state and goal state (e.g., Rubik's Cube), the distinction between forward and backward search is impossible.

Backward search has been applied to various games or their endgames (Chess, Checkers, Awari, Nine Men's Morris) and to parts of the 15-Puzzle state space. Gasser [28] has implemented a general workbench (the Search-Bench) for sequential retrograde analysis, gives references to parallel backward search results and suggests that retrograde analysis is easier to parallelize than forward search because a parallel forward search algorithm may be forced to expand states that a sequential algorithm might prune.

3.1.4 Combination of Backward and Forward Search

Some authors have successfully combined the advantages of forward and backward search. Gasser [28] solved Nine Men's Morris by such a combination. He first computed the values of about one-half of the positions in the game by backward search and then determined the value of the initial position by alpha-beta (forward) search. He also implemented a branch-and-bound algorithm for the 15-Puzzle which uses a database of relaxed subproblems computed by backward search as lower bounds. A similar strategy let us solve the NUG25 instance of the quadratic assignment problem in parallel [10]. Some authors have made experiments with parallel bidirectional search applied to the 15-Puzzle [47, 65].

3.1.5 Heuristic Search

As all known algorithms for the exact solution of NP-complete optimization problems in the worst case use exponential time, and practical problems most often are much larger than those that can be solved in a reasonable amount of time, people have invented techniques which trade solution quality for the time needed to find solutions. These techniques are called *heuristic methods, metaheuristics* or *local search*—the terminology is not uniform [42, 52, 1]. In general, they visit a random subset of the state space and return the best solution encountered without a guarantee for its quality. The more time available, the more states can be searched and the higher should be the quality of the solution. Although most authors base their work on mystic analogies to physical and biological processes and prefer practical experiments to theoretical reasoning, they all report good results. However, none of the heuristic methods can prove anything about the quality of the solutions found. In this section, we classify the main heuristic search methods and investigate what has been done for their parallelization [49].

Heuristic search algorithms interpret the feasible solutions of an optimization problem as the vertices of a graph (state space) by defining an adjacency relation (*neighbors* of a state).[1] The adjacency relation should satisfy the following three requirements:

- The state space should be connected. Otherwise, we won't find good solutions if we start the search in the wrong component.
- Neighboring solutions should have correlated objective functions. Although this is a vague requirement, we can understand it intuitively. The algorithms walk along the edges of the graph from good solutions to better solutions. If the objective functions of adjacent states were totally independent, we could as well dispense with adjacency and just generate completely random states.
- There should be few local optima which are not global optima because the main challenge for heuristic algorithms is avoiding to get stuck in a local optimum. Ideally all local optima should be globally optimal.[2]

Heuristic search methods are cheaper to implement than branch-and-bound algorithms. For both approaches the implementation of a skeleton doesn't pose major problems. In exhaustive search, however, the bounding functions can be rather complex (e.g., 1-tree for the TSP or polyhedral methods for other problems), whereas programs for heuristic methods essentially consist only of a neighborhood function.

We classify heuristic search methods into the following classes:

1. **Simulated Annealing**
   Simulated annealing is a simple technique which tries to avoid local optima. It uses a terminology borrowed from physics. The objective function of a state is called its *energy,* and the main parameter of the algorithm is called *temperature.* The algorithm does a random walk on the state space. Beginning in a starting state, it repeatedly selects a random neighbor, computes the energy difference of the old and the

[1] We use the terms *solution* and *state* interchangeably in this section.

[2] The Simplex algorithm (although it is not a heuristic search algorithm) shows us that even this strong requirement is not a sufficient condition for polynomial complexity.

new state, and accepts the new state depending on the result of the comparison. A new state of lower energy is always accepted. A new state of higher energy is accepted with a probability P depending on the energy difference

$$P = e^{-\Delta E/T}.$$

It follows that at high temperatures every new state is accepted; at low temperatures a new state is accepted only if this operation decreases the energy. The algorithm starts with a high temperature and proceeds according to a *cooling schedule* until the temperature is so low that the random walk stops at a local minimum. The choice of the cooling schedule has a high influence on the convergence properties of the algorithm and its efficiency.

Parallelization of simulated annealing is difficult because a random walk is an inherently sequential process. One approach (speculative computation) follows several possible paths in parallel while the energy of the first new state is evaluated. If the new state is accepted, the paths which do not include it are discarded. If it is not accepted, the other paths (which include the state) are discarded. The achievable speedup is low; it depends on the acceptance rate of new states, which itself depends on the temperature. If the principle of having a single random walk is given up and the parallel algorithm is no longer required to compute exactly the same path as sequential simulated annealing, higher speedups are possible [21]. However, the physical analogy suggests that it is more profitable to slowly grow one good crystal instead of quickly cooling ten bad ones.

Another (application-dependent) approach partitions the data of original problem and tries to anneal the parts in parallel [35].

2. **Tabu Search**
   Tabu search [31] is another kind of random walk. It tries to avoid becoming stuck in a local optimum by keeping a tabu list of states or directions which have been visited and must be avoided in the near future. In every step it moves from a state to the best neighbor which is not in the tabu list.

   If the neighborhood of a state is big enough and the computation of values slow enough, we can parallelize tabu search by distributing the neighbors onto the available processors [49]. Another, less communication-intensive, type of parallelization lets every processor compute its own path through the space.

3. **Methods Based on Biological Evolution**
   Unlike the techniques introduced above, evolution-like methods (e.g., genetic algorithms, evolution strategies, evolutionary programming) do not work on a single solution of the optimization problem, but rather on a collection (a set with multiple elements) of them, which is called *population.* They simulate a biological model based on the three principles of crossover, mutation and survival of the fittest:

   - Pairs of population members generate offspring whose genetic information consists of a random combination of the parents' chromosomes. For the implementation of this idea, we need a *crossover* operator which generates children which resemble both parents, that is, which can combine good parts of both. Defining a good crossover operator is much more difficult than defining a good neighborhood for simulated annealing or tabu search. Whereas it is easy to imagine a crossover of two covers in the vertex cover problem, it is difficult to find a reasonable crossover of two traveling salesman tours.
   - To avoid premature convergence of the algorithm and to preserve population diversity, a random *mutation* operator is necessary. This operator corresponds to the neighborhood of the other methods.
   - By crossover and mutation, new solutions are generated. A selection step selects the fittest of them, while the rest is removed from the population. The *fitness* is defined by the value of the objective function.

   A straightforward parallelization of genetic algorithms [43] distributes the population onto the processors (subpopulations) and computes the crossover and mutation operations, as well as the fitness values in parallel. For the selection of the fittest individuals, synchronization is necessary. In the island model, the subpopulations evolve separately on islands, and synchronization is reduced to migration phases. Individuals either migrate from any island to any other, or only among neighboring islands. The latter approach avoids global synchronization. The isolation of the islands can lead to higher population diversity and better results compared to a single population. Some authors attribute this effect to the parallel computer instead of the model and call it superlinear speedup (instead of implementing the island model sequentially).

4. **Greedy Randomized Adaptive Search Procedure (GRASP)**
   GRASP consists of two phases: a construction phase and a local search phase. In the construction phase, an initial reasonable solution is constructed at random, and this solution is taken as a starting point for finding a local optimum in the second phase. These two phases are repeated; the best solution found is kept and gives the final result. As the iterations of this loop are completely independent, GRASP is easily parallelized [49].

5. **Other Methods**
   A wealth of other heuristic methods has been invented (e.g., neural networks, threshold algorithms, dynamic hill climbing). In addition to that, almost any technique of the above can be combined with any other to form hybrid methods. Most of them have not yet been parallelized.

An extensive bibliography on heuristic search methods can be found in [42]. There is a wealth of articles describing the good results achieved by applying some method to some problem, but only few authors compare the efficiency of different methods.

Although ZRAM does not yet contain parallel search engines for heuristic algorithms, Frank Ammeter has implemented a parallel GRASP algorithm for the vehicle routing problem on top of the ZRAM load balancer, proving that ZRAM is suited for the inclusion of parallel heuristic search. In fact, such search engines might be a valuable addition to the library.

3.2 Models and Tools

Parallel computing has introduced a whole bunch of new issues into computer science. These issues range from theoretical questions (what algorithms are parallelizable and to what extent?) to practical problems (what is a good user interface for a parallel debugger?). Many branches of computer science address these issues. In this section, we summarize what has been done to assist both theoreticians and software developers. We further show where ZRAM fits in this context and which of the available tools helped in ZRAM's development. It turns out that theoreticians use only machine models, whereas software developers apply a wide range of tools.

To discuss theoretical questions, we need *models* of parallel computers. A machine model abstracts of real machines and tries to capture their essential characteristics. As sequential models, the Turing machine and the random access machine (RAM) are widely used. The most important characteristic of a parallel machine, its ability to perform several operations in parallel, has been captured in the PRAM model, which resembles a RAM, but executes several operations in parallel in every time step. It is well suited for the design of parallel algorithms, but its shared memory does not model the communication costs of real parallel computers.

Another approach was made by Hoare with his communicating sequential processes (CSP) model, which applies to message-passing algorithms and emphasizes their correctness rather than their efficiency [34].

To overcome the deficiencies of the PRAM, the community has developed models such as the bulk-synchronous parallel (BSP) and the LogP model [19]. They both have a distributed memory where processors communicate by point-to-point messages. The LogP model specifies the performance characteristics of the interconnection network, but does not describe the structure of the network. Its name comes from the four parameters that describe the machine: L (latency), o (overhead per message), g (gap, which is the time per transferred data unit or the reciprocal of the bandwidth), and P (number of processor–memory modules). Many other models have been published. It has even been said that “the number of parallel computer models exceeds the number of different parallel computer architectures that have been either proposed or built” [54].

During the development of ZRAM, we got to the conclusion that all these models are much more important for theoretical reasoning than for software development; they are necessary for proving theorems on the complexity of algorithms, but neglect the constant factors which are so important in practice—who develops programs on the Turing machine?

Much research has been carried out on *parallel programming languages.* The researchers in this field strive to define high-level notations which help the programmer in developing correct and efficient parallel programs. They often restrict the class of algorithms which can be expressed, take the control over the generated code away from the programmer and move the responsibility to the compiler—the analogy to sequential high-level languages is obvious.

Most of these languages have been implemented on very few machines, and many of them are unsuitable for the irregular parallel search algorithms which are of interest to us. To ensure portability, we have chosen to implement ZRAM in ANSI C rather than in a parallel programming language, for C is the only programming language available on all the computers we used.

Parallelizing compilers have attracted a lot of interest, mainly because there are millions of lines of old Fortran code around and people hope to parallelize this code without rewriting it manually. These compilers are able to distribute data which is stored in arrays onto the local memories and to distribute regular computations onto processors, generating the necessary data movement operations automatically. In some cases, the programmer has to help the compiler by inserting compiler directives into the code, but this approach is still cheaper than rewriting whole programs. Parallelizing compilers are less general than their proponents claim them to be. They are good for old programs with a regular structure which can be parallelized in a straightforward way. For irregular algorithms, however, human insight can—in our opinion—not be replaced by a compiler; these algorithms should rather be provided to other users in the form of libraries, as we provide search algorithms in the ZRAM library.

A wealth of other tools is supposed to help in the development of parallel programs. Most important for the development of ZRAM was MPI (Message Passing Interface) [33], a library of point-to-point and collective communication which is available on a broad range of parallel computers. A similar library is PVM (Parallel Virtual Machine) [29]. In the future, we expect other libraries to appear which provide more complex functions.

As a direct assistance to the programming process, there exist parallel debuggers and performance monitors. Performance monitors show, often graphically, where in a computation processors are idle (waiting for messages) and resources are wasted. In the development of ZRAM, we used the performance-monitoring software PARAGRAPH.

3.3 Software Libraries

In the modern world of computer science, much manpower (and money) is wasted by people duplicating programs. This can happen when they ignore that another implementation exists, when they have no access to the other implementation, or when they do have a copy of the source code, but it is badly structured, mixed with other parts of the application, or has the wrong interfaces.

A library is the principal tool for reusing code. When a library is built, a complicated task is solved once, but the solution is used several times. The users of a library, who use it as a black box, save time by not having to acquire particular knowledge about the field, and by avoiding to implement, test and maintain the code themselves.

There is the prejudice of library code being less efficient than special-purpose code. In practice, this is rarely a problem: The savings in programmer time far outweigh the cost of the additional execution time. Measurements with ZRAM have shown that the execution time typically increases only by a few percent over optimized code (see Section 7.7).

Libraries exist in many fields of computer science, such as computational geometry (e.g., GeoBench, CGAL, LEDA), graphical user interfaces (e.g., X Windows), data bases, linear algebra (e.g., BLAS, LINPACK), optimization (e.g., CPLEX, ABACUS) and parallel programming (e.g., PVM, MPI). These fields have three characteristics in common:

1. There is a large number of applications which can use the library. It would be unreasonable to develop a library for which only one application exists.

2. The algorithms hidden in the library are complex enough that a programmer who uses the library really saves time. The time for learning the interface must be smaller than the time necessary the implement the algorithms.

3. There is some consensus on what algorithms and data structures should be in the library, and the interfaces are reasonably standardized.

The field of parallel search algorithms certainly fulfills the first two criteria. First, there are many applications waiting for the power of parallel computers. Second, parallel programming is still a difficult task, as in parallel programs arise various issues unknown in sequential programming (e.g., nondeterminism, deadlock, load balancing, termination detection). The third criterion (standardization of interfaces) is usually achieved only after several libraries have been implemented and compared. Parallel search algorithms are in this phase now: There are several libraries to compare (see Figure 3.1).

Experience shows that every library must find the right compromise between generality and specialization. If a library is too general, its interface becomes large and difficult to use, and its implementation unmanageable and difficult to maintain. If a library is too specialized, potential users will be forced to implement another variant of the code themselves. PPBB puts the main weight on load-balancing algorithms, BOB on data structures for branch and bound. ZRAM is more general in the diversity of the implemented algorithms. It shows that different parallel search algorithms have enough common parts to justify their being in the same library. On the other hand, ZRAM is specialized to message-passing computers. It uses another, lower-level library (MPI), which ensures portability even to shared-memory computers.

3.4 Performance Measures

To evaluate the performance of sequential programs, we measure execution time and memory usage as a function of the input size. Parallel computing introduces the number of processors as an additional independent variable, making it impossible to present the behavior of an algorithm completely in the two-dimensional graph of a function $t = f(n)$. Instead of that, we define the following performance metrics as abstractions of the complete data:

- **Speedup**
  (Real) speedup for a fixed input is defined as the execution time of the fastest sequential algorithm divided by the execution time of a parallel algorithm, computed as a function of the number of processors. It answers the question: "How much faster is my program with more processors?" This measure does not depend on the absolute speed of the computer; it depends only on the ratio of communication to computation speed, the program, and the chosen input. In practice, the fastest sequential algorithm is often replaced by the parallel algorithm executed on only one processor *(relative speedup)*. Speedup is called *ideal* or *linear* if it is equal to the number of processors and *superlinear* if it is higher.

  Although speedup often is easily measured, some difficulties are associated with the concept:

  1. The fastest sequential algorithm is often unknown or expensive to implement.
  2. Speedup depends on the input (size). Good speedup for a given input does not imply that a different input gives the same speedup.
  3. Speedup doesn't say much about absolute execution time.
  4. Measuring speedup is impossible for an input which is too large to be processed on a single processor.
  5. Speedup typically favors slow computations (code that is inefficient when run on a single processor) over fast ones.
  6. In areas such as heuristic search, the goal of parallelization is often improving the quality of the approximate solution rather than shortening the execution time.

- **Efficiency**
  Efficiency is defined as the ratio of speedup and the number of processors. It answers the question: "What percentage of the parallel computer is being used productively?"

- **Isoefficiency**
  Isoefficiency (sometimes called *scalability*) is defined as the rate at which workload must increase relative to the rate at which the number of processors increases so that the efficiency of the parallel system remains the same. It answers the question: "How much larger problems can I solve with more processors?" Whereas it overcomes some of the difficulties of speedup, measuring isoefficiency implies finding an input resulting in a given efficiency on a given number of processors. In particular for irregular search algorithms, this task is almost impossible.

An overview of these and other performance metrics together with their merits and problems can be found in [54]. Time measurements on a parallel computer have a larger variance than measurements of sequential programs because of the inherent nondeterminism and the high complexity of a parallel operating system, which both affect the execution time in unpredictable ways. We show speedup and isoefficiency measurements in Chapter 7.

3.5 Hardware

A given program does not run on every computer, even if it is called *portable.* On certain systems it runs, but it runs inefficiently. A library designer therefore has to decide early in the development process which computer architectures the software will support. To this end, we give a short classification of parallel machines and explain the design decisions which affect the portability of ZRAM:

1. **Control Mechanism**
   With respect to the number of control units, parallel machines are classified either as SIMD (single instruction multiple data) or MIMD (multiple instruction multiple data). A SIMD machine has many arithmetic-logic units (ALU), but only one control unit. In every time step, all ALUs execute the same instruction. In a MIMD machine, in contrast, which has as many control units as ALUs, all processors work independently. SIMD machines were invented in order to save the cost of the control units, as in a parallel program often all processors execute the same algorithm, but on different data. Today, mostly MIMD machines are built from standard microprocessors for economic reasons: Special-purpose ALUs can never be produced in the same number of pieces as a standard microprocessor. ZRAM has been designed for MIMD computers.

2. **Memory Architecture**
   An architecture where every processor has its own (local) memory and cannot access the other memories directly is called a *distributed-memory* or *message-passing* architecture because the processors communicate by passing messages. The contrary is a *shared-memory* architecture where all processors access the same (global) memory. There are various intermediate forms such as the Intel Paragon MP where the processors are grouped into pairs which share their memory, but communicate with the other pairs by sending messages. Another intermediate form is *virtual shared memory* (also called *distributed shared memory*), where the operating system simulates shared memory on top of distributed-memory hardware by fetching memory pages from another processor when a page fault occurs. ZRAM has been written for message-passing machines for two reasons. First, we think that a distributed-memory architecture is best suited for massively parallel machines. Second, most shared-memory machines provide some message-passing library, but a program written for shared-memory rarely runs on distributed-memory machines.

3. **Topology**
   The designer of a distributed-memory machine has to choose a method of connecting the processors. In small machines, all processors can be attached to a single bus. In larger machines, point-to-point links are more common. The graph of these links often takes the form of a grid (Paragon), torus or hypercube (Cenju-3). ZRAM ignores the topology of the machine because dependence on the topology decreases portability and because our applications are so coarse-grained that the topology has little effect on the performance.

4. **Number of Processors**
   The size of parallel machines varies greatly: By definition, a parallel computer has at least two processors, but so-called *massively parallel* machines with up to 65536 processors have been built (e.g., the Connection Machine CM-2). These four orders of magnitude are not only a quantitative change, they also induce a qualitative change in programming methods. ZRAM has been used with up to 150 processors.

5. **Communication Speed**
   The performance of sequential computers is measured in instructions per second (or floating-point operations per second, depending on the focus of the user). Parallelism adds another dimension, namely the speed of communication. Often, a linear dependence between message

length and communication time is assumed. In this model, the communication speed is defined by two parameters:

- *latency:* the time taken by an empty message (measured in seconds)
- *bandwidth:* the additional data transferred per time unit (measured in bytes per second)

A derived quantity is the ratio of communication to computation speed. It is defined as the time to communicate a standard message (e.g., one floating-point number) divided by the time for one computation operation (e.g., one floating-point multiplication). Depending on this ratio, a machine is suitable for either coarse-grain or fine-grain parallelism. Examples can be found in [53].

ZRAM has been used on a Paragon, a Cenju-3, a GigaBooster, a Sun Ultra, and workstation networks.

Chapter 4

Design of ZRAM

We now draw the general picture of ZRAM and so prepare the ground for the three subsequent chapters (Chapters 5–7) where the different layers are presented separately. We first state our requirements for a parallel search library and develop the principles which guided the design of ZRAM (Sec. 4.1). Then we present the architecture of ZRAM as well as the design decisions of interest and discuss all issues which are common to several layers of ZRAM (Sec. 4.2).

4.1 Requirements for a Parallel Search Workbench

We have identified five requirements for a parallel search workbench: generality, simplicity, usability, efficiency, and portability. In the following paragraphs, we illustrate these requirements, show the conflicts among them, explain their influence onto ZRAM's design, and show how we can judge whether ZRAM has attained the stated goals.

4.1.1 Generality and Flexibility

A library must have a certain generality and flexibility to be useful. However, this goal conflicts with the aim for efficiency and simplicity, and ZRAM is the result of our endeavor to find a balance between these objectives. A more general alternative to our library of parallel search algorithms might have been a library of parallel graph algorithms or of parallel combinatorial algorithms. A less general alternative would have been a library of parallel branch-and-bound algorithms such as BOB.

An impression of ZRAM's flexibility is given in Chapter 7, where we present its broad range of applications.

4.1.2 Simplicity

Simplicity is not a design goal for itself, but it helps to achieve usability, efficiency and portability. Moreover, it leads to reliable, easily testable code. We have made ZRAM as simple as possible while providing a system of practical use.

4.1.3 Usability

Software which is difficult to use wastes the time of its users, makes them angry, leads to high support expenses, or is not used at all. Because of these facts we made usability a design goal for ZRAM. A measure for usability is the time which users must invest until they have a productive result [48].

Usability conflicts with the goal of generality. Systems that want to be general tend to have procedures with long parameter lists and many options that must be understood before the system can be used productively. In this conflict, we have favored usability.

Three factors contribute the most to the usability of ZRAM:

Layered structure ZRAM has a layered structure with clear interfaces between layers (see Figure 4.1). The higher an interface is in this structure, the higher is its abstraction level. The programmer can choose which interface to use. Whereas standard applications can be built on the existing search engines, which hide the complexity of parallel programming completely, programmers desiring more flexibility can access the virtual machine directly.

Simplicity The search engines are tailored to specific algorithms and provide only the necessary options.

Uniformity of interfaces The interfaces of all search engines have a uniform structure using the same two data types and similar application-defined functions.

The people who used ZRAM to parallelize their own applications have confirmed that it is user-friendly (Sec. 7.7). They all learnt to use it within one to three days. This is a good result, all the more because half of them had no prior experience with parallel computing at all.

4.1.4 Efficiency

There is only one reason for using parallel computers: speed. Why should we use parallel computers unless to get a result quicker than with a sequential computer? Hence, a parallel search library must be efficient.

We can show the efficiency of a library either by comparing it to other software, or by solving large benchmark problems. The direct comparison approach is cumbersome, as we would have to procure comparable software and to provide similar conditions to both competitors. This method can imply porting the software to a new computer and implementing the same application on top of both libraries—a huge and boring task.

We have chosen the second approach and solved large instances of problems which are interesting for themselves and which could not be solved before. These results are described in Chapter 7.

4.1.5 Portability

Portability is a standard design requirement in software development. It is compulsory if software is to outlast a rapidly changing environment. Whereas there is a high degree of standardization in sequential computing, parallel computing is still dominated by a wide range of incompatible systems with a short lifetime. The ZRAM project was begun on a Transputer card in a Macintosh. Neither this Transputer card nor our second parallel computer, MUSIC with its MUSIC operating system have survived until today. The project would have been impossible unless portability was one of the main goals since the beginning.

One design choice affected by the strive for portability is the programming language used. The only language available on all the systems we have used is ANSI C. C++ might have led to better-structured code, but good code is useless if there is no compiler available to compile it.

Software cannot be portable to every conceivable system. Early in the design process, the designer has to restrict the range of systems considered. For ZRAM with its coarse-grained algorithms, we have decided to concentrate on message-passing MIMD computers and to exclude SIMD machines. The decision for message-passing is no real restriction because messages can be emulated on shared-memory machines. Furthermore, we achieve topology-independence for a small penalty in efficiency by simply disregarding the communication topology of the particular machines.

The ZRAM architecture follows a usual approach toward portability. It contains a thin host-system layer (see Section 5.5) in which almost all machine dependencies are hidden.

4.2 Architecture of ZRAM

4.2.1 Three Interfaces and Four Layers

ZRAM's architecture is based on three interfaces that separate four layers of software (Figure 4.1).

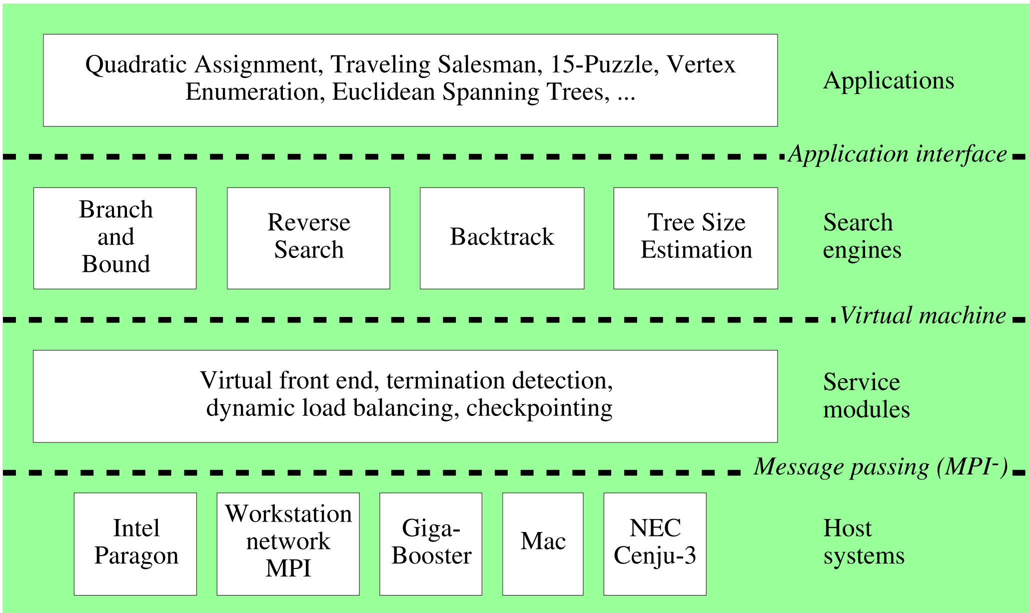

Figure 4.1: The ZRAM architecture. The programmer of the top layer is not concerned with parallelism, as all parallel constructs are hidden in the lower three layers. All layers except the bottom one are machine-independent.

- **Message Passing Interface**
  This interface makes ZRAM portable by hiding all machine dependencies of the host systems. Because it has been modeled on a subset of the standard message passing interface MPI, we call it *MPI*$^-$. It retains the basic nonblocking point-to-point send and receive primitives of MPI, but omits collective communication, communicators, etc. MPI$^-$ 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.

- **Application Interface**
  Most important for typical users of ZRAM is the application interface, as they will program directly on top of this interface, which has two functions: It separates the problem-specific parts of a program from the problem-independent parts, and it confines the explicit parallelism in the layers below. A library of search engines that run on top of the virtual machine makes up the layer that implements this top-most interface. Search algorithms (and the data structures they use) implemented so far include backtrack, branch-and-bound, reverse search [4], and a tree-size estimator based on sampling [39].

Could we work with fewer than three interfaces? No. The message passing interface is necessary for portability, the application interface for application independence, and without the virtual machine all search engines would have to implement the common services themselves. The three interfaces divide ZRAM into four layers, which determine the structure of this thesis. The topmost three layers are described in separate chapters (the application layer in Chapter 7, the search engines together with the application interface in Chapter 6, the service modules together with the virtual machine interface in Chapter 5), and the small host-system layer at the bottom is described in Sec. 5.5.

That the machine-dependent host-system layer makes up only an eighth of the ZRAM code (Figure 4.2) demonstrates that the goal of a portable library has been achieved. The pie chart also shows that the virtual machine interface divides the middle two layers into two equal-sized parts (2900 lines each), the ratio which structures the code most effectively. The application layer has been excluded from the calculation because its size varies with the number of applications added, and because it has nothing to do with the size relation of the lower three layers.

The interface structure of other search libraries is less sophisticated than the one of ZRAM. Typically, the designers of these libraries had the objective of developing one or more branch-and-bound (BOB, PUBB) or load-balancing (PPBB) engines which could be reused for several applications, putting less emphasis on other search algorithms and on portability. In all cases, this led to the definition of an application interface, but no (at least no published) virtual machine interface.

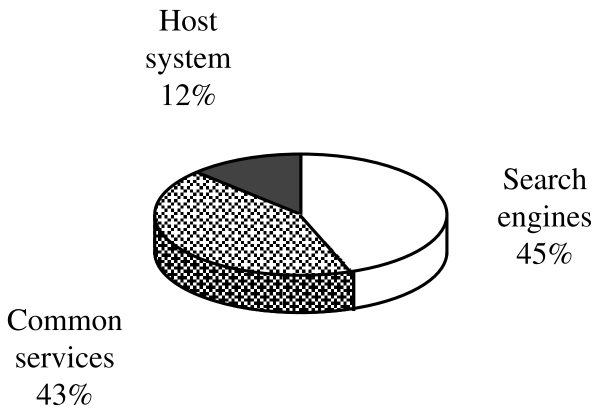

Figure 4.2: Relative size of the three application-independent layers of ZRAM. The host-system layer has been kept as small as possible to ensure high portability.

4.2.2 Extensibility

ZRAM claims to be an extensible library. In what directions can it be extended? Probably not vertically by adding a fifth layer, but by extending single layers horizontally. Typical ZRAM users extend it by adding new applications to the top layer. This task is easy because it requires only basic knowledge of parallel computers.

The host-system layer covers the two most important groups of systems: single-processor machines and MPI systems. The sequential machines are perfect for developing and debugging new applications, and the MPI standard is supported by most distributed-memory multiprocessors. Although many shared-memory machines provide the same message-passing interface (MPI), a new implementation of the host-system layer for shared memory is possible and might be more efficient.

The search-engine layer can be extended through the addition of more search engines. Most interesting would be a parallel branch-and-cut engine (see Sec. 8.2).

Extending the common services is possible, but should be demand-driven: Potential improvements to the virtual machine will become apparent during the implementation of further search engines.

4.2.3 Skeletons and Upcalls

ZRAM provides not only library functions, but also algorithm *skeletons* in the form of search engines. The distinctive property here is that a skeleton calls application-specific functions programmed by the user, whereas a library function calls only other library functions. When a skeleton calls an application-specific function, which is implemented in a higher software layer than the skeleton, this call is denoted as *upcall.*

For many areas (e.g., linear algebra) library functions are flexible enough. A general branch-and-bound system designed as a set of library functions supplies the user with procedures for subproblem-pool management (e.g., insert subproblem, get highest-priority subproblem). However, parallel search algorithms clearly profit from the additional flexibility of skeletons. A search engine is most naturally expressed as a skeleton calling user-defined functions.

When a programmer writes a C program and wants to use a library function, there usually exists a header file, and the programmer knows the name of the function being called. We have the same situation in ZRAM when a procedure in a higher layer calls a lower-layer procedure (e.g., the call to `z_bt_Enumerate` in Figure 2.2). In an upcall (e.g., the call to `nq_BranchInPlace` in Figure 2.2), the situation is different. When the search engine is written, the names of the branching functions which will be written in the future are unknown.

The issue can be resolved if we define a fixed name for all branching functions, use this name in the search-engine code and have the linker bind the correct procedure to the name. To our knowledge, all branch-and-bound libraries use this simple and efficient approach. However, this approach restricts flexibility: In a single statically linked program, the search engine can be used for only one purpose; the program cannot have more than one branching function, because at runtime the linker cannot change the function called by the search engine.

ZRAM uses a different, less simple but equally efficient and more powerful approach. A ZRAM application defines a *procedure set,* a data structure containing pointers to the relevant procedures. It then assigns a number to the procedure set by calling `z_install_procset` and supplies this number to the search engine as a parameter. The additional level of indirection created by assigning numbers to procedure sets is necessary because (procedure) pointers are not valid across processors in a distributed-memory architecture.

The same approach is used at the virtual machine interface when the common services call procedures pertaining to a search engine.

4.3 Data Types

A search tree always consists of nodes, which have a data type (in a programming language that knows data types). Although the data in a tree node depends on the application, the search-algorithm layer and the common services make some assumptions about it. In this section we describe these assumptions, how they would be enforced in an object-oriented language and how we implemented them in C.

In an object-oriented language such as C++, the lower layer of the library defines a base class, and the application defines a derived class. The library would define

```
class z_Node { // base class
  long len;     // length in bytes; used to pack the node
                // into a message
  // other data used by the search engine
};
```

and a traveling salesman application would define

```
class tsp_Node: public z_Node { // derived from z_Node
  // data describing a partial tour
};
```

As ZRAM is written in C, which is not an object-oriented language, we need another method to define the interface to the application. ZRAM uses the same approach as other search libraries, namely the definition of header fields which must be included in every search tree node. Thus, the library defines (in z_base.h)

```
#define Z_NODE_HEADER uint32 len;              \
                      uint16 tag;              \
                      uint16 children;         \
                      /* some more fields */
```

and the traveling salesman application defines

```
struct tsp_Node {
  Z_NODE_HEADER; /* header fields defined in z_base.h */
  /* data describing a partial tour */
};
```

The `len` and `children` fields are used by the message-passing system to pack nodes into a message. In other parallel search libraries, these fields do not exist; the user has to supply pack and unpack procedures for every data type to make message passing possible. The task of writing these procedures is straightforward but boring; they would be necessary if we decided to port ZRAM to a heterogeneous system, but add only little flexibility in our current environment.

Moreover the header contains a `tag` field, which is used for debugging, and room for pointers. The reverse-search engine uses these pointers to maintain the ancestor cache (see Section 6.3.2). All objects must be allocated by `z_NewNode()` rather than `malloc()` so that their headers are initialized correctly.

In every ZRAM application, the user has to define two data types, one for the global data and one for the tree nodes. In this regard, ZRAM differs from branch-and-bound libraries such as PUBB (three types) and BOB (two types, but not the same ones as ZRAM). Both PUBB and BOB use different data structures for regular tree nodes and solution nodes. This distinction can save memory because some information in the regular tree nodes may be redundant if it is known that the node is a solution. One the other hand, the small number of solution nodes that are generated during a computation limits the achievable memory saving, and ZRAM permits the programmer to use different layouts for regular and solution nodes. BOB seems to work without having a data type for the global variables.

4.3.1 Compression of Nodes

Search tree nodes often contain a huge amount of redundant information in order to speed up a computation. In the vertex enumeration application (Sec. 7.1), a full dictionary is stored (md multi-precision integers for a d-dimensional polyhedron with m facets) although m bits designating the facets in the current basis would give the full information. In the vertex cover program (Sec. 7.3), the degree of every vertex of the remaining graph is stored although one bit per vertex would suffice.

Sometimes, it is desirable to have two representations of nodes, one which allows for a fast computation by keeping redundant data and a memory-efficient one. The redundant representation would be used for most computations, whereas the other (memory-efficient) one would be used to keep messages and checkpoint files short, and to store nodes in the priority queue of best-first branch-and-bound. The latter use is of particular importance, as the range of problem instances that can be solved by a branch-and-bound program is limited by the number of nodes which fit in the priority queue (the size of which is determined by the available memory).

The current version of ZRAM does not support short and long representations of nodes, but this enhancement is simple to implement in such a way that the user has only to provide a compress and an uncompress function.

4.3.2 Incremental Storage

A tree node in ZRAM, together with the global variables, always completely defines a state of the search space. In applications such as the 15-Puzzle, an alternative representation is possible. Instead of storing the full 4×4-board, we could store a pointer to the preceding state and the move (north, east, south, west) which led to the current position. ZRAM does not contain any provisions for such a data structure, and adding one would be a major task.

Chapter 5

Virtual Machine and Common Services

The virtual machine interface is the core of ZRAM. It divides ZRAM into two halves, defining the functions which are provided by the bottom half and used by the top half. It is the most important interface, being used by every search algorithm in the library. Only the virtual machine justifies collecting several search algorithms to form a single library. Without it, every search engine could as well stand alone.

5.1 Requirements for a Virtual Machine

Hiding machine dependencies by defining a virtual machine is a time-honored technique to achieve portability. 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.

In our aim for a simple system, we have found that three functions are essential for parallel searching:

- dynamic load balancing

- checkpointing

- termination detection

In addition, our virtual machine provides a virtual front end, message-passing operations, as well as the two abstract data types priority queue and sequence.

5.2 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 five 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 5.3.3) and one specialized to best-first search (speculative priority queue, see Section 5.3.4).

2. **Checkpointing**
   Our computations take days or weeks on a powerful parallel computer. Because it is rarely possible to get that much processor time in one chunk, a checkpointing facility is necessary (see Section 5.3.3). Checkpointing means saving the state of a computation in external memory so that the computation can be interrupted and resumed. 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 for global termination detection (see Section 5.4.1).

4. **(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 (Figure 5.1). Every call to a search engine on the front end corresponds to one iteration of the loop by the slaves (see Section 5.3.1). In contrast to a pure master–slave model, the processing elements (slaves) may communicate with one another during the compute phase by using MPI's point-to-point communication. We thus distinguish *front-end communication* and *horizontal communication* (Figure 5.2). 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.

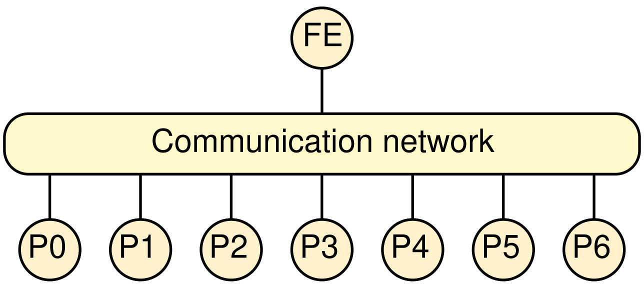

Figure 5.1: ZRAM's process model. In ZRAM, not all processors are equal. One of them is distinguished as the virtual front end and serves for the sequential parts of the computation. Computers such as the CM-2 or the MUSIC are based on this model.

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 disadvantages (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 (see Section 5.3.2).

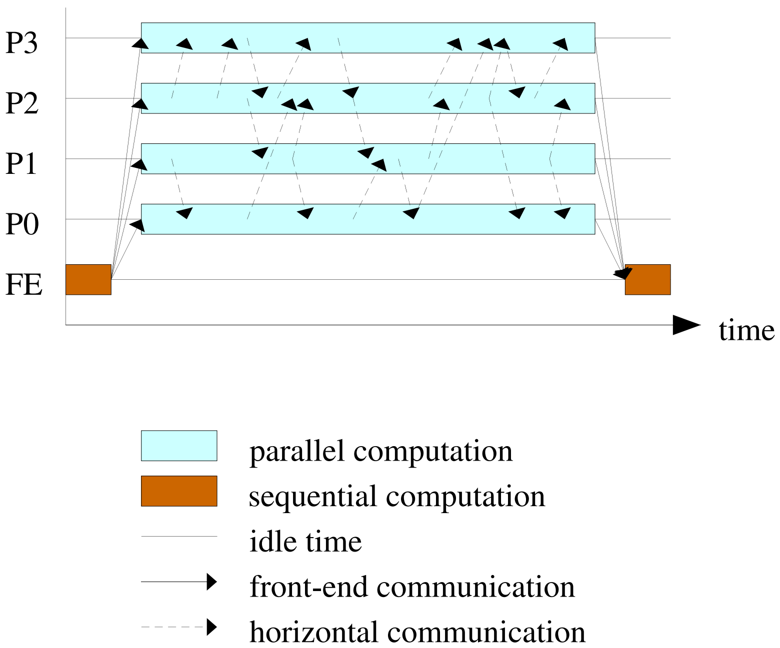

Figure 5.2: Execution of a ZRAM program in time. The front end executes the sequential parts of the computation and initiates a parallel job; that is, the main program calls a search engine. In most of our applications, the sequential part takes little time (seconds), whereas the parallel part can take weeks to complete. The implementation maps the front end and P0 to the same processor.

5.3 Interface

In this section, we describe the virtual machine interface—that is, the main functions building the interface between the search engine layer and the common service layer. We begin with the communication functions (separated into front-end and horizontal communication), and continue with the load-balancing interface. We omit the fairly standard interface to the two abstract data types priority queue and sequence as well as various functions of minor importance.

5.3.1 Virtual Front End

ZRAM distinguishes one processor as a virtual front end which executes the sequential parts of the computation (the application-defined `mmain()` function), and it treats communication between the front end and the other processors (front-end communication) differently from communication during the parallel computation (horizontal communication, see Figure 5.2). Our virtual machine provides three functions for the purpose of front-end communication. The first one, `z_InstallJob`, is to be called by the initialization code of all processors, whereas the other two functions are called exclusively on the front end.

```
z_JobNum z_InstallJob(const z_VoidProc job);
```

The initialization code (`initproc()`) calls `z_InstallJob` to associate a procedure with a number. The given procedure will later be executed on all processors in parallel. It should begin by reading the input buffer and end by writing a result to the output buffer.

```
void z_StartParallelJob(z_JobNum jobnum,
                        z_ResultProc resultProc);
```

The front end calls `z_StartParallelJob()` to initiate a parallel computation. It broadcasts the contents of the output buffer, and then all processors execute the procedure associated to `jobnum`. At the end of the parallel computation, every processor sends a message to the front end, and this message is read by `resultProc()`.

```
void z_WaitWork(void);
```

`z_WaitWork()`, called on the front end, waits until the parallel computation initiated by `z_StartParallelJob()` completes.

With hindsight, we can see that the separation of `z_StartParallelJob()` and `z_WaitWork()` offers unnecessary flexibility. Although this design makes it possible that the front end computes something while the other processors execute a parallel job, none of our applications profits of that flexibility—they all call `z_WaitWork()` immediately after `z_StartParallelJob()`.

5.3.2 Horizontal Communication: Message Passing with Termination Detection

MPI is a complex communication library (its interface contains more than a hundred routines); we use only a few of them. But it lacks two features which are important for our applications. First, it does not provide a mechanism to detect the termination of a distributed program. Second, it obstructs information hiding and modular software development. During the execution of a branch-and-bound program, a processor can receive two kinds of messages: load-balancing messages and new upper bounds. Whereas we want to process the former in the load balancer, the latter should be routed to the search engine. If we use plain MPI, some point of our program code must contain a complete list of the possible messages:

```
MPI_receive(message);
switch (message) {
  case load_balancing:
    ...
  case new_upper_bound:
    ...
}
```

ZRAM's virtual machine defines a tiny message-passing module which resolves the two problems mentioned. Our module comprises the functions listed in Table 5.1, as well as functions for writing objects of type `z_Node` to the output buffer and reading them from the input buffer.

Function	Description
`z_Isend`	sends the output buffer to another processor
`z_Ibroadcast`	sends the output buffer to all other processors
`z_InstallMessage`	defines a function to be called when a message of a certain type is received
`z_IdleWithHandler`	waits until a message is received or until all processors are idle
`z_Poll`	tests whether a message is received

Table 5.1: Message-passing functions in ZRAM. All these functions interact with the termination detection algorithm.

5.3.3 General Load Balancing and Checkpointing

ZRAM contains a general load-balancing and checkpointing module. This module manages a global queue of *work units*. A work unit may be any object describing the work to be done (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). We require the application or the search engine (i.e., the module which uses the load-balancing and checkpointing module) to define the two operations `Work` and `Split` on work units (Figure 5.3). Checkpointing amounts to saving the global queue of work units to a file.

Functions Defined by the Application or Search Engine

```
void Work(z_lb_GlobalP global,
          z_lb_WorkUnitP *workUnitH)
```

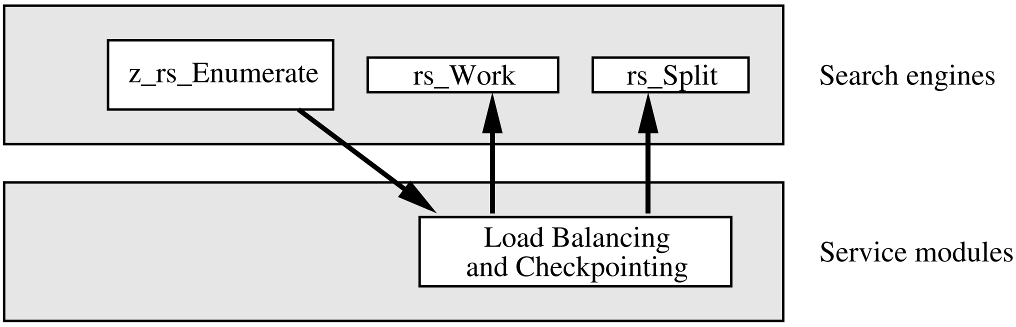

Figure 5.3: Call graph of load balancing and checkpointing. The reverse-search engine defines the functions `Work` and `Split`. `Work` does the real work, whereas `Split` splits work units so that a part of the work can be transferred to another processor. After the search engine has inserted an initial work unit into the global queue, the load-balancing and checkpointing module calls `Work` and `Split` as needed.

`Work` receives a work unit describing a task to be done. It executes a part of the task and returns a new work unit describing the work which still remains to be done.

```
void Split(z_lb_GlobalP global,
           z_lb_WorkUnitP workUnit,
           s_seqP workList)
```

`Split` receives a work unit describing a task to be done. It splits this task into subtasks and inserts the resulting work units into the list `workList`. A `Split` function which always simply inserts the work unit received into the list (i.e., splits the task trivially into one subtask), leads to a correct result, but prevents any load-balancing functionality.

Instead of providing `Work` and `Split` separately, the user can choose to combine them into a single function `WorkAndSplit`. This option is useful if splitting the work implies executing part of it.

```
void WorkAndSplit(z_lb_GlobalP global,
                  z_lb_WorkUnitP workUnit,
                  s_seqP workList)
```

`WorkAndSplit` executes a part of the work described by `workUnit`, splits the rest into subtasks and inserts the resulting work units into the list `workList`.

Calls to the Virtual Machine

The application or the search engine starts the load-balancing and checkpointing module by a call to `z_lb_Work`. An interrupted computation is continued by a call to `z_lb_RestartWork`.

```
void z_lb_Work(z_ProcSetNum procsetnum,
               z_lb_GlobalP global,
               z_lb_WorkUnitP workUnit,
               boolean sequential)
```

`z_lb_Work` broadcasts the global data in `global` to all the processors and calls `Work` and `Split` repeatedly until all the work in `workUnit` is done.

```
void z_lb_RestartWork(z_ProcSetNum procsetnum,
                      z_lb_GlobalP global,
                      boolean sequential)
```

`z_lb_RestartWork` reads a checkpoint file written by `z_lb_Work` and continues the interrupted computation at that point. `global` must be identical to the `global` given to `z_lb_Work` when the checkpoint was written.

```
void z_lb_InstallCheckpointed(z_NodeP *nodeh)
```

By a call to `z_lb_InstallCheckpointed`, an application or a search engine tells the system that the object pointed to by `nodeh` is to be written to every checkpoint file (in addition to the queue of work units, which is always written). This feature, which can be used for every object that should be checkpointed, is used by the branch-and-bound engine to make the current upper bound a part of the stored data.

5.3.4 Special Load Balancing: The Speculative Priority Queue

We have implemented two variants of sequential and parallel branch-and-bound engines: *depth-first,* and *best-first.* Depth-first, which requires a stack, balances computational load by using the load-balancing module described in Section 5.3.3. Best-first, which requires a priority queue, is approximated by a search engine 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. It is a trade-off between the communication overhead involved in finding a global minimum and the search overhead caused by the expansion of nonminimum nodes.

The speculative priority queue of ZRAM is an abstract data type providing the following operations:

```
z_spqP z_spq_NewSpeculativePriorityQueue(z_pq_CompProcP compare)
```

```
void z_spq_FreeSpeculativePriorityQueue(z_spqP pq)
```

These two operations create and destroy speculative priority queues. They must be called on all processors at the same time. The ordering on the queue elements is defined by the comparison function given as a parameter when the speculative priority queue is created.

```
void z_spq_Insert(z_spqP pq,
                  z_NodeP x)
```

This operation inserts an element `x` into the speculative priority queue.

```
z_NodeP z_spq_GetDeleteNearMinimum(z_spqP pq)
```

This operation retrieves an element near the minimum and deletes it in an atomic operation. It returns `NULL` if the speculative queue is empty (this test uses the termination detection algorithm). In order to decouple the processors, we do not guarantee that the returned element is the minimum. The test whether the queue is empty, however, is exact.

```
long z_spq_numLocalElements(z_spqP pq)
```

`z_spq_numLocalElements` returns the number of elements of `pq` which are stored in local memory. Because of the dynamic load balancing, the value returned may change between successive calls.

```
void z_spq_DeleteLocalAbove(z_spqP pq,
                            z_NodeP upperBound,
                            whatToDoProcP whatToDo)
```

`z_spq_DeleteLocalAbove` removes all elements of `pq` which are in local memory and greater than or equal to `upperBound` from the priority queue and calls `whatToDo` for each of them. `whatToDo` must not access the priority queue, but should for instance deallocate the elements.

5.4 Implementation

In this section, we present the most interesting observations in the implementation of the common services—that is, in the software layer between the virtual machine interface and the MPI^- interface. These observations occur in the dynamic load-balancing algorithms and in the checkpointing facility.

5.4.1 Termination Detection

ZRAM contains a standard algorithm [45] for global termination detection. Every processor counts the messages it receives and the messages it sends. A single token, which is sent around when processors are idle, keeps information on the state of every processor and serves to compute the total number of messages sent and received. When the processor which has the token detects that all processors are idle and all messages sent have been received, it broadcasts this information to all other processors.

5.4.2 Load Balancing

ZRAM's dynamic load-balancing mechanism (relaxed queue) is currently used by all search engines except best-first branch-and-bound. Because 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*. 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 lots of new nodes, all processors are usually busy working off the initial node assigned to them. Only toward the very end of the computation, 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, we have decided to select the third and all succeeding processors in a round-robin fashion rather than randomly.

5.4.3 Checkpointing

Where is the ideal place to implement checkpointing? Data transfer between internal and external memory resembles the transfer of data among processors; in both cases, for instance, data structures are linearized. Writing data to disk corresponds to sending a message, reading from disk corresponds to receiving a message, and a disk file corresponds to a message. For these reasons, we have made ZRAM's checkpointing an integral part of the load-balancing module.

At regular intervals (e.g., every hour), the virtual machine interrupts the computation, brings the dynamic load-balancing algorithm into a known state by synchronizing all processors, and saves the global queue of work units to disk. It then saves other relevant data as defined by the application, such as current bounds or the sum of the determinants in the computation of polytope volumes. On machines supporting a signal facility, the operator also can trigger checkpointing by sending a signal to the process group.

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.

To provide against the loss of all data which might occur if the machine crashes while the checkpoint files are written, ZRAM optionally makes a back-up copy of the old files before the new ones are written and identifies the files with a generation number.

5.4.4 Speculative Priority Queue

The implementation of the speculative priority queue uses a local priority queue on every processor and a load-balancing mechanism which exchanges elements among these local queues. With the load-balancing mechanism, we pursue two different goals:

- We want to avoid processors becoming idle when their local queues are empty although the global queue still contains elements. For this purpose, the load-balancing mechanism tries to equalize the size of the local queues by spreading size information and by transferring elements to queues which have fewer elements than others.
- We want the function `z_spq_GetDeleteNearMinimum` always to return elements near to the global minimum. For this purpose, the load-balancing mechanism simply shuffles elements around.

The load-balancing mechanism generates two kinds of messages: *node* messages, which transfer elements and size information, and *empty* messages, with which a processor signals that its local queue is empty. A processor receiving a *node* message of a processor which owns fewer elements than itself responds by sending back another *node* message containing a number of elements proportional to the size difference of the queues. A processor receiving an *empty* message either responds with a *node* message, or forwards the *empty* message, or ignores it. These responses are not generated immediately when the message is received, but only during the succeeding `z_spq_GetDeleteNearMinimum` operation.

Almost all operations on the speculative priority queue are implemented locally. `z_spq_GetDeleteNearMinimum` is the only operation which can initiate communication. It distinguishes two cases:

1. If the local priority queue is not empty, `z_spq_GetDeleteNearMinimum` retrieves the minimal element and initiates a load-balancing operation; that is, it sends the size of the local queue and some elements to another processor, either spontaneously or in response to a message received earlier.

2. If the local priority queue is empty, `z_spq_GetDeleteNearMinimum` sends an *empty* message and waits until either a *node* message which contains elements arrives or global termination is detected.

Because of the complexity of this algorithm, we summarize the three steps of the correctness proof (indicating that `z_spq_GetDeleteNearMinimum` returns an element when the global queue is not empty and correctly detects when the global queue is empty):

1. The algorithm is *partially correct* (i.e., if it terminates, the result is correct): An element which has been inserted into the global queue can only be removed by `z_spq_GetDeleteNearMinimum` or `z_spq_DeleteLocalAbove`. The algorithm terminates only when all local queues are empty and there are no messages sent but not yet received.

2. *Deadlock* is impossible: A processor waits for messages only when its local queue is empty. If all processors wait, the algorithm terminates.

3. There is *progress:* It is impossible that all processors indefinitely send messages around rather than computing because a processor sends a bounded number of messages only when one of two conditions is met: (1) its queue is empty or (2) it has just removed an element from the queue (and thus made progress).

Our experiments with large search applications have shown that a more sophisticated and fine-tuned load-balancing algorithm is not necessary. When the search tree is large enough, any algorithm seems to be able to balance the load sufficiently well.

5.5 Host-System Layer

At the bottom of ZRAM lies a thin host-system layer (see Figure 4.1). It serves as a central place for gathering most of the machine-dependent code to ensure the portability of the other three layers. It provides the basic message-passing and time-measurement primitives in a machine-independent form. The message-passing operations are organized in two groups:

Front-end communication The front end starts a job by a broadcast, and after the completion of the job, it collects the result data.

Horizontal communication There are three functions for sending messages during the execution of a job (immediate send, immediate broadcast, and test whether the send has completed) and three functions for receiving messages (immediate receive, test whether a message has been received, and wait until a messages is received). There is no provision for termination detection at this level.

Three instances of the host-system layer have been implemented:

mpi The mpi instance uses the MPI library [33]. It runs on workstation networks, the NEC Cenju-3, the GigaBooster, and the Sun Ultra.

nx The nx instance uses the Intel NX library and runs exclusively on the Paragon. It was the first implementation of the host-system layer and has strongly influenced the design of its interface.

seq This instance runs on single-processor machines such as the Macintosh and Unix workstations. It simulates the communication between the front end and a single processing element on one processor. Its horizontal message-sending procedures must never be called, the message-receiving procedures never return a message, and termination detection is trivial.

Time measurement is a function which differs from computer to computer; it is incompatible even among the various Unix dialects. ZRAM provides a unified interface through the following function:

```
double z_Gtime(void)
```

`z_Gtime` returns a time value in seconds. On parallel computers, it measures elapsed time (wall clock time) because we want to include idle time in the measurement; on sequential systems, it returns the CPU time consumed by the process because we do not want to include CPU time used on behalf of other processes.

Chapter 6

Search Engines

The parts of ZRAM most visible to the user are its search engines. In this chapter we show what they can be used for, how they are accessed, how they are implemented, and we discuss the relevant design choices. We begin by describing what all the search engines have in common, and then devote a section to every search paradigm.

ZRAM's search engines are implementations of the three general-purpose search paradigms backtrack, branch-and-bound, and reverse search. Other algorithms (e.g., branch-and-cut, alpha-beta, GRASP), could be added. Because branch-and-bound is a general paradigm and admits many technical variations, it is implemented by five distinct search engines.

The three paradigms backtrack, branch-and-bound, and reverse search all solve problems by computations on trees. They differ in the method used to define the tree, in the traversal of the tree (depth-first, best-first or variants of them), and in the computations done during the traversal.

There is no explicit parallelism at the application interface level. All the search engines share the same style of interface. The user of a search engine 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 which are to be called by the search engine (Figure 6.1)

Both the global data and the node are abstract data types for the search engines; the engines access them only through application-defined operations. 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

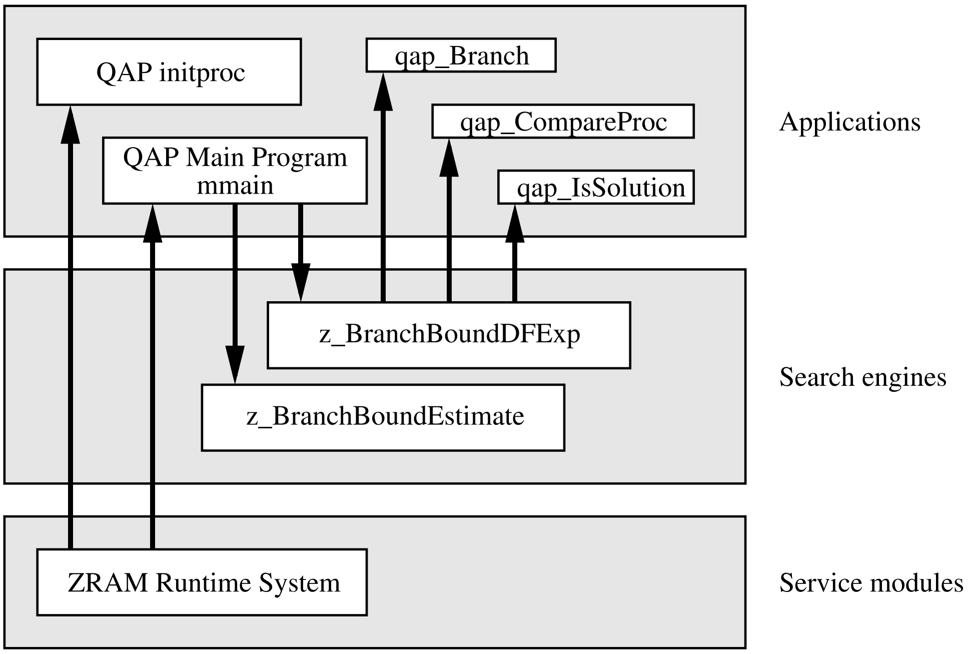

Figure 6.1: Call graph of a branch-and-bound application (Quadratic Assignment Problem) with respect to ZRAM's layered structure. The application's main program calls a search engine, and the search engine calls application-dependent functions. The interfaces to the backtrack and reverse-search engines are similar.

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, where coarse granularity does not limit achievable speedup [9]. The tree-size estimator calls the same problem-specific routines as the search engines.

The implementation of the search engines relies on ZRAM's virtual machine. Every engine either applies the load balancer by defining `Work` and `Split` operations or uses the speculative priority queue. Whereas no parallelism is visible to the application layer, the only explicit parallel operation of the search layer is the broadcasting of upper bounds by the branch-and-bound engines. The architecture is extensible to further search engines.

6.1 Backtrack

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 to prune subtrees. This restriction simplifies the interface (no comparison between nodes) and the implementation (no broadcasting of upper bounds). The application has to provide only a branching procedure, and the search engine takes care of the rest.

6.1.1 Interface

ZRAM requires a backtrack application to provide one application-defined function, whereas it provides a choice of two search engines.

Application-Defined Functions

A backtrack application defines a single function (see Section 2.2 for an example):

```
boolean appl_BranchInPlace(appl_GlobalP global,
                           appl_NodeP n,
                           z_bt_DoProc whatToDo,
                           void *ref)
```

`BranchInPlace` determines the children of `n` and calls `whatToDo(n, ref)` once for every child. After the final call to `whatToDo`, it restores the information in `n`. `whatToDo` can call `BranchInPlace` recursively or send a copy of `n` to another processor for balancing the parallel load. `BranchInPlace` must return 0 (nonzero values are reserved for future extensions).

Search-Engine Calls

There are two backtracking search engines. Both of them require as parameters a pointer to the `BranchInPlace` function, the global data, and the root of the search tree.

```
z_NodeP z_bt_EnumerateDepthFirst(z_ProcSetNum procsetnum,
                                 z_bt_GlobalP global,
                                 z_NodeP root)
```

This search engine enumerates the whole search tree sequentially by recursive calls to `BranchInPlace`. It is optimized for execution on a single processor and outperforms the other search engine in this case.

```
z_NodeP z_bt_EnumerateDFExp(z_ProcSetNum procsetnum,
                            z_bt_GlobalP global,
                            z_NodeP root,
                            boolean sequential)
```

`z_bt_EnumerateDFExp` is the parallel backtrack engine. If the parameter `sequential` is 0, it executes in parallel; otherwise it executes sequentially.

Discussion

ZRAM provides a `whatToDo` procedure to the branching function, rather than requiring the branching function to return a list of the subproblems generated. We explain this design choice below in the discussion of the branch-and-bound interface (Section. 6.2.1).

6.1.2 Implementation

`z_bt_EnumerateDepthFirst` is a simple, but extremely efficient search engine. Its `whatToDo` function simply calls `BranchInPlace` recursively.

`z_bt_EnumerateDFExp` uses the load balancer of ZRAM's virtual machine (see Figure 6.2). It is a simplified version of `z_BranchBoundDFExp` (see Section 6.2.1): It does not store an upper bound and does not cut off subtrees.

Discussion

When we compare the execution time of the two backtrack engines applied to the n-queens problem with its short and efficient branching function, it turns out that `z_bt_EnumerateDFExp` running on one processor is three times slower than `z_bt_EnumerateDepthFirst`. This overhead, which would be less noticeable in an application which has a longer computation time per node, is produced mainly by the memory management functions of the C library. Whereas `z_bt_EnumerateDepthFirst` neither allocates or deallocates memory nor copies data, `z_bt_EnumerateDFExp` allocates a memory block for every node of the search tree, copies the data generated by `BranchInPlace`, and inserts the block into a list. It would be interesting to combine the advantages of both engines.

6.2 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

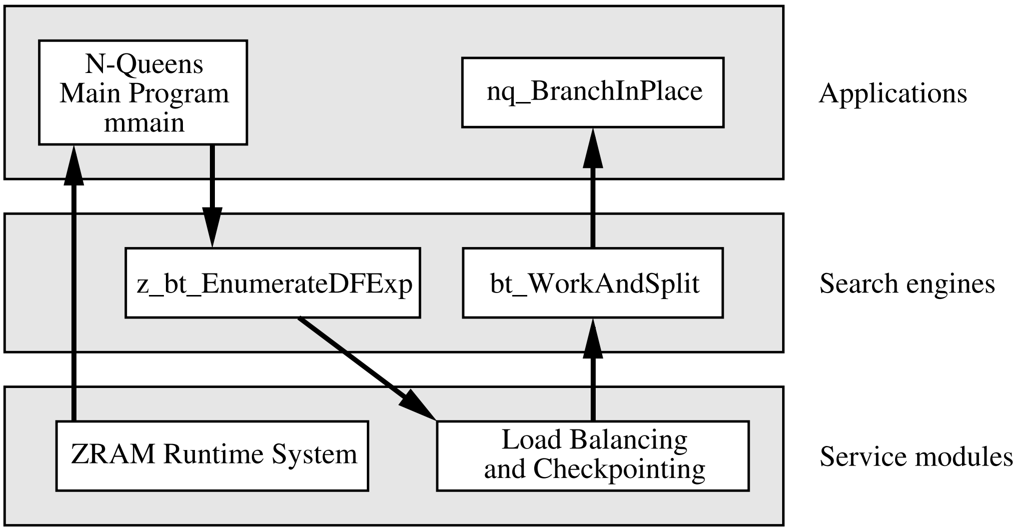

Figure 6.2: Implementation of the parallel backtrack engine. The search-engine layer calls the load balancer of the layer below and provides it a `WorkAndSplit` function. The `WorkAndSplit` function in turn calls the application-specific `BranchInPlace` function and returns the list of the children to the virtual machine.

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 it generates the smallest search trees possible, but it can only be used if enough memory is available.

6.2.1 Interface

Every branch-and-bound search engine in ZRAM needs three application-dependent functions: a branching function, which splits a problem into subproblems and computes lower bounds; a solution test; and a function which compares the lower bounds stored in two nodes.

All the definitions of this section can be found in the file z_bb.h. The file bb_example.c contains a small example program which demonstrates the use of the branch-and-bound interface.

Application-Defined Functions

Every branch-and-bound application defines at least three functions: `Compare`, `IsSolution`, and either `Branch` or `BranchInPlace`.

```
void appl_Branch(appl_GlobalP global,
                 appl_NodeP n,
                 appl_NodeP upperBound,
                 z_bb_DoProcP whatToDo,
                 void *ref)
```

`Branch` is the main function of a branch-and-bound application. It determines the children (subproblems) of `n`, allocates memory for them (by calling `z_NewNode`), computes their lower bounds and calls `whatToDo(child, ref)` for each of them. It can read the current best solution (`upperBound`), which may be useful in deciding how much time to invest in the computation of the lower bounds. It does not deallocate the memory for the children—this is the responsibility of the search engine. `Branch` can be called recursively by `whatToDo`.

```
void appl_BranchInPlace(appl_GlobalP global,
                        appl_NodeP np,
                        appl_NodeP upperBound,
                        z_bb_DoProcP whatToDo,
                        void *ref)
```

`BranchInPlace` is an alternative to `Branch` and more efficient for depth-first search. Instead of allocating memory for the children of a node, it overwrites the parent node for each child and restores the parent after the final call to `whatToDo`. The search engine never deallocates the memory for a child computed by `BranchInPlace`. `whatToDo` can call `BranchInPlace` recursively (depth-first), store a copy of the child in a priority queue (best-first) or send the child to another processor (load balancing).

```
int appl_CompareProc(appl_NodeP x,
                     appl_NodeP y)
```

This function compares the bounds stored in two nodes. It returns 1 if $x > y$, 0 if $x = y$, and -1 if $x < y$.

```
boolean appl_IsSolution(appl_NodeP n)
```

This function returns 1 if the subproblem described by `n` is a solution of the original problem and 0 otherwise.

A branch-and-bound application may define two optional functions: `UpperBound` and `ShortNodeString`.

```
appl_NodeP appl_UpperBound(appl_GlobalP global,
                           appl_NodeP n)
```

`UpperBound` heuristically computes an upper bound for the solution of a subproblem `n`. If the function is available, the search engine calls it before the search algorithm is executed. In a depth-first search, a good upper bound makes early cutoffs possible and keeps the search tree small.

```
char *appl_ShortNodeString(appl_NodeP n)
```

`ShortNodeString` returns a pointer to a statically allocated string describing the node `n`. It is used by the search engine to generate debugging output.

Search-Engine Calls

ZRAM provides various branch-and-bound engines. There are three sequential engines (depth-first, best-first, and a combination thereof), and two parallel ones (parallel best-first, and parallel restartable depth-first). As parameters, they all expect a pointer to the application-defined functions in `procsetnum`, the global data in `global`, the root of the search tree in `root`, a flag `noisy` indicating whether debugging output is desired, and optionally a feasible solution for cutting off parts of the tree in `initialUpperBound`. If `initialUpperBound` is `NULL`, but the `UpperBound` function is defined, all the search engines call this function to compute a feasible solution.

Every search engine stops as soon as it has found one solution and proved that it is optimal, returning this optimal solution, or it returns `NULL` if the problem has no feasible solution.

```
z_NodeP z_BranchBoundDepthFirst(z_ProcSetNum procsetnum,
                                z_bb_GlobalDataP global,
                                z_NodeP root,
                                z_NodeP initialUpperBound,
                                int noisy)
```

This engine executes a sequential depth-first search, calling either `Branch` or `BranchInPlace` recursively.

```
z_NodeP z_BranchBoundBestFirst(z_ProcSetNum procsetnum,
                               z_bb_GlobalDataP global,
                               z_NodeP root,
                               z_NodeP initialUpperBound,
                               int noisy)
```

This engine does a sequential best-first search, storing the nodes of the tree in a priority queue. As long as the queue fits into the available memory, this strategy generates the smallest possible search trees. However, if there is insufficient memory for the priority queue, the program crashes.

```
z_NodeP z_BranchBoundMixedFirst(z_ProcSetNum procsetnum,
                                z_bb_GlobalDataP global,
                                z_NodeP root,
                                z_NodeP initialUpperBound,
                                int queueSize,
                                int noisy)
```

This engine combines the advantages of best-first and depth-first search. It executes a best-first search as long as there are fewer than `queueSize` nodes in the priority queue. When the queue attains a size of `queueSize`, a depth-first search is made. This strategy takes advantage of best-first search as long as possible, but avoids crashes of the program because of insufficient memory. `queueSize` must be set by the application such that the priority queue and the deepest possible depth-first stack fit into the available memory together.

```
z_NodeP z_BranchBoundParallel(z_ProcSetNum procsetnum,
                              z_bb_GlobalDataP global,
                              z_NodeP root,
                              z_NodeP initialUpperBound,
                              int queueSize,
                              boolean noisy)
```

`z_BranchBoundParallel` is the parallel version of the mixed-first engine. Allocating a global speculative priority queue with `queueSize` elements per processor, it searches best-first as long as it can insert elements into the queue, and depth-first when the queue overflows. This search engine should be used whenever there is no good initial upper bound available and checkpointing is not desired.

```
z_NodeP z_BranchBoundDFExp(z_ProcSetNum procsetnum,
                           z_bb_GlobalDataP global,
                           z_NodeP root,
                           z_NodeP initialUpperBound,
                           boolean noisy,
                           boolean sequential)
```

This procedure performs a depth-first search with checkpointing. If `sequential` is 1, the computation is done sequentially, if it is 0, the search is done in parallel. This search engine should be used when a good upper bound is known, or when checkpointing is desired.

```
z_NodeP z_RestartBranchBoundDFExp(z_ProcSetNum procsetnum,
                                  z_bb_GlobalDataP global,
                                  boolean sequential)
```

This procedure restarts an interrupted branch-and-bound computation at the checkpoint saved in the checkpoint file. All three parameters must be given identical values as in the original call to the search engine.

```
int z_bb_RedistributeDFExp(z_ProcSetNum procsetnum,
                           int n_in,
                           int n_out)
```

If we have started a parallel branch-and-bound computation on a given number of processors and want to continue it using more or fewer processors, a checkpoint must be written and processed by `z_bb_RedistributeDFExp`. This procedure reads the checkpoint files created by `n_in` processors and writes them back as if they were created by `n_out` processors. It prints the statistical data collected and reinitializes it to zero. If the operation was successful, `z_bb_RedistributeDFExp` returns zero. Otherwise it returns an error number.

Discussion

There is an alternative to requiring an application-specific `Compare` procedure: We could require that the node data type contains a value field of a fixed type (e.g., `double`) which can be accessed by the search engine. Other branch-and-bound libraries follow this approach. We claim that the ZRAM approach is more flexible, because there might be optimization problems over a totally ordered set which cannot be mapped to `double` so easily, and because the application designer may want to evaluate a lower bound only when it is needed in a comparison, and only to the precision necessary to determine the outcome of the comparison.

Instead of providing a `whatToDo` procedure to the branching function, other branch-and-bound libraries require that the branching function returns a list of the subproblems generated. Either method can be emulated by the other one. The method used in ZRAM has the advantage that a memory-efficient recursive depth-first search can be implemented which uses only space linear in the depth of the tree, rather than the depth of the tree multiplied by the branching factor. The same consideration applies to the backtrack interface.

Why does `z_bb_RedistributeDFExp` exist only for branch-and-bound and why is it not part of the virtual machine? A checkpoint consists of two components, the work units (subproblems to be solved), and the current upper bound of every processor (the upper bounds may differ from one another if the checkpoint is written after a processor finds a new upper bound, but before the corresponding broadcast). The former can be handled by the virtual machine, but the latter are specific to branch-and-bound.

6.2.2 Implementation

A parallel branch-and-bound engine operating in distributed memory cannot save an upper bound (best solution found so far) in a global variable as a sequential engine does. In the two parallel engines (`z_BranchBoundParallel` and `z_BranchBoundDFExp`), every processor maintains a local upper bound (the best solution *it* currently knows). Whenever the local upper bound decreases, its new value is immediately broadcast to all other processors. When a processor receives a message containing an upper bound, it compares the bound in the message to the local one and keeps the stronger one. This strategy ensures that eventually every processor knows the globally best solution. If messages are delayed, weaker bounds are used, fewer subtrees cut off and more nodes evaluated than if they arrive quickly. Such delay generates search overhead, but does not affect the correctness of the algorithm.

`z_BranchBoundParallel` approximates a best-first search by using the speculative priority queue which is provided by the virtual machine. It repeatedly fetches one of the first elements of the queue (those with the best lower bounds) and calls `Branch` or `BranchInPlace`. `whatToDo`, which is called for every subproblem generated, selects the first applicable choice of the following list:

1. If the lower bound of the new subproblem is at least as high as (equal to or worse than) the current upper bound, the subproblem is simply

deallocated and forgotten.

2. If the new subproblem is a solution of the original problem, it replaces the old upper bound, is stored and broadcast to all processors. All queue elements whose lower bound is higher than the new upper bound are removed.

3. Otherwise, the new subproblem is inserted into the speculative priority queue.

`z_BranchBoundParallel` does not make checkpoints of the speculative priority queue. Because this queue tends to fill up all the memory available, the checkpoint files would be extremely large.

`z_BranchBoundDFExp` uses the load balancer of the virtual machine. The work units of the load balancer are the subproblems of the search engine. The load balancer calls `bb_WorkAndSplit` which in turn calls `Branch` or `BranchInPlace` (compare Figure 6.2). `whatToDo`, which is called for every subproblem generated, selects the first applicable choice of the following list:

1. If the lower bound of the new subproblem is at least as high as the current upper bound, the subproblem is simply deallocated and forgotten.

2. If the new subproblem is a solution of the original problem, it replaces the old upper bound, is stored and broadcast to all processors. All queue elements whose lower bound is higher than the new upper bound are removed.

3. Otherwise, the new subproblem is appended to the end of the list of work units maintained by the load balancer. Because the load balancer always selects the last element of the list for evaluation, this strategy amounts to a depth-first search.

The checkpointing facility, which is implemented in the load balancer, saves not only the work units, but the current upper bound also.

6.3 Reverse Search

Reverse search [3, 4] is a memory-efficient technique for traversing a graph without marking visited vertices. Its time complexity is linear in the output size and its space complexity does not depend on 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 defined 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.

6.3.1 Interface

ZRAM's reverse-search engine needs five application-dependent functions: the local search function, the adjacency oracle, an equality test for vertices, a root test, and the maximum degree in the graph.

ZRAM's reverse-search component is the first parallel implementation of reverse-search. Its interface conforms exactly to the standard defined in [4]. An overview of it is given in Figure 6.3, and the formal definition can be found in the file z_rs.h. The file rs_example.c contains a small example program which demonstrates the use of the reverse-search interface by enumerating the points of a finite rectangular grid.

Application-Defined Functions

A reverse-search application always defines the five mandatory functions `F` (local search), `AdjOracle` (adjacency oracle), `Equal` (equality test), `IsRoot` (root test) and `Delta` (bound for the number of adjacencies):

```
appl_NodeP appl_F(appl_GlobalP global,
                  appl_NodeP node)
```

This is the local search function [4]. Given any node of the search tree which is not the root, this function computes its parent.

```
appl_NodeP appl_AdjOracle(appl_GlobalP global,
                          appl_NodeP v,
                          uint32 k)
```

The adjacency oracle defines the underlying graph structure. Given a node `v` and a number k with $0 \leq k < \delta$, it returns a node adjacent to `v` or `NULL`.

```
uint32 appl_Delta(appl_GlobalP global,
                  appl_NodeP node)
```

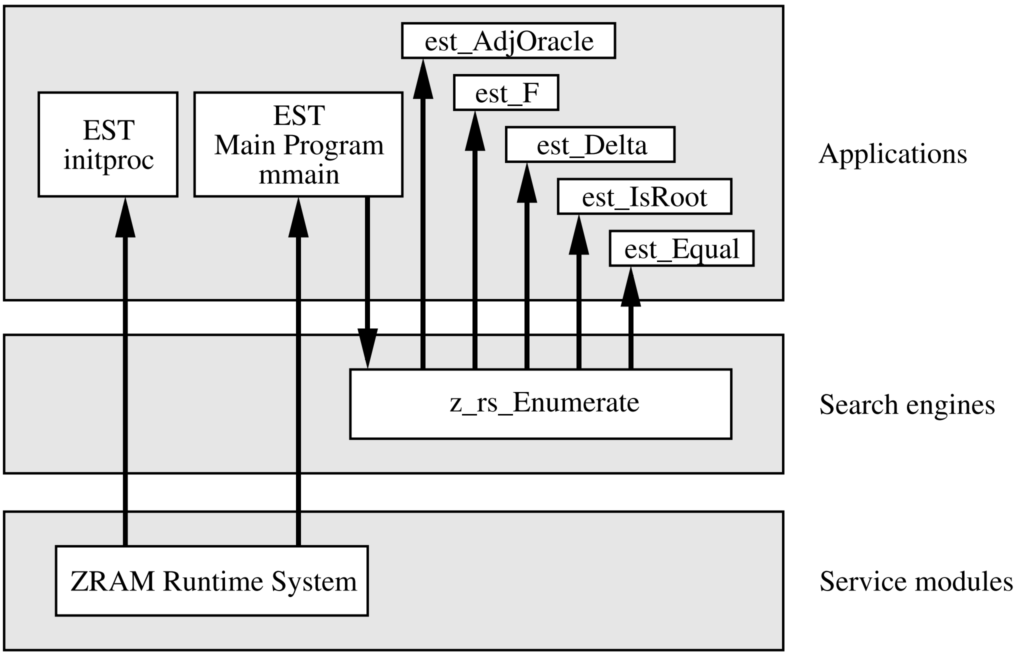

Figure 6.3: Call graph of a reverse-search application (enumeration of Euclidean spanning trees) with respect to ZRAM's layered structure. This application defines only the five mandatory functions. Other applications (e.g., vertex enumeration) define further optional functions.

This function returns an upper bound on the degree of a given node.

```
boolean appl_Equal(appl_GlobalP global,
                   appl_NodeP a,
                   appl_NodeP b)
```

This function tests two nodes for equality. It returns 1 if they are equal and 0 if they are not.

```
appl_IsRoot(appl_GlobalP global,
            appl_NodeP node)
```

This function returns 1 if the given node is the root of the search tree and 0 if it is not.

The application may define optional functions. Three of them (`FindChild-Number`, `Forward` and `NextChild`) can enhance the efficiency of the computation, and the fourth one (`Result`) is usually used to print the enumerated objects.

```
uint32 appl_FindChildNumber(appl_GlobalP global,
                            appl_NodeP child,
                            appl_NodeP parent)
```

This is the inverse function to the adjacency oracle. It returns the first (and unique) integer k such that `AdjOracle(parent, k)` = `child`. If it is defined, it replaces the following code fragment:

```
for (k=0;; k++) {
  other = AdjOracle(global, parent, k);
  if (other) {
    if (Equal(global, other, child)) break;
    z_FreeNode(other);
  }
} /* for */
z_FreeNode(other);
```

```
void appl_Forward(appl_GlobalP global,
                  appl_NodeP *nodeH,
                  uint32 *k)
```

In certain applications (e.g., vertex enumeration), it is possible to determine the child number in the computation of the local search function without calling the adjacency oracle. In these cases, `Forward` can define a more efficient computation replacing the following three-line code fragment of the reverse-search engine:

```
node = F(global, child);
k = FindChildNumber(rs_global, child, node);
z_FreeNode(child);
```

```
appl_NodeP appl_NextChild(appl_GlobalP global,
                          appl_NodeP parent,
                          int k,
                          int l,
                          uint32 *num)
```

This function can replace the following code fragment which tries to find a child with a number between `k` and `l` and returns this number in `num`:

```
step = (l > k) ? 1 : -1;
for (; k != l; k += step) {
  next = AdjOracle(global, parent, k);
```

```
    if (next) {
      if (!IsRoot(global, next)) {
        f_next = F(global, next);
        if (Equal(global, parent, f_next)) {
          /* found a child */
          z_FreeNode(f_next);
          *num = k;
          return next;
        } /* if */
        z_FreeNode(f_next);
      } /* if */
      z_FreeNode(next);
    } /* if */
  } /* for */
  return NULL; /* there are no more children */
```

```
void appl_Result(appl_GlobalP global,
                 appl_NodeP node)
```

This procedure (if defined) is called by the search engine for every node enumerated. It can, for instance, print the node.

Search-Engine Calls

```
void z_rs_Enumerate(z_ProcSetNum procsetnum,
                    z_lb_GlobalP global,
                    z_NodeP root,
                    boolean sequential)
```

`z_rs_Enumerate` is the main call to the reverse-search engine. It enumerates nodes beginning at `root`. The read-only global data is given in `global`, and the application-specific functions are given in `procsetnum`. If `sequential` is 1, the enumeration is done sequentially, if it is 0, the tree is enumerated in parallel. The computation is checkpointed as described in Section 5.2.

```
void z_rs_Restart(z_ProcSetNum procsetnum,
                  z_lb_GlobalP global,
                  boolean sequential)
```

This procedure restarts an interrupted reverse-search enumeration at the checkpoint saved in the checkpoint file. All three parameters must be given identical values as in the original call to the search engine.

Discussion

Although the mandatory part of the interface seems to be concise and clear, we could eliminate the function `IsRoot`, leaving only four mandatory functions. We see two possibilities for this elimination. First, we could replace `IsRoot` by an equality test, for the search engine knows the root node. This solution is not satisfactory because the root node would have to be broadcast to all processors and because the equality test may be more expensive than the root test.

A more efficient solution is modifying the definition of the local search function `F` such that it returns `NULL` when it is called for the root (the current definition does not allow the root node as a parameter to `F`). This modification would suffice because the only place where the root test is used in the implementation is just before the call to `F` in `NextChild`. One might think that another root test is necessary to detect the termination of the reverse-search algorithm, but the search engine always knows the depth of the current node in the tree and stops when this depth decreases to zero.

6.3.2 Implementation

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. After ZRAM had shown that this idea can double the speed of an enumeration, David Avis integrated the same concept into his vertex enumeration program `lrs`. Load balancing transfers subproblems (intervals) among processors, but not the ancestor cache. This loss of context contributes to the parallelization overhead.

The memory requirements of reverse search do not depend on the size of the graph, and we can save any intermediate state of the sequential computation 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, and therefore has to define the work units as well as the work and split operations in terms of the five application-specific functions (Figure 6.4).

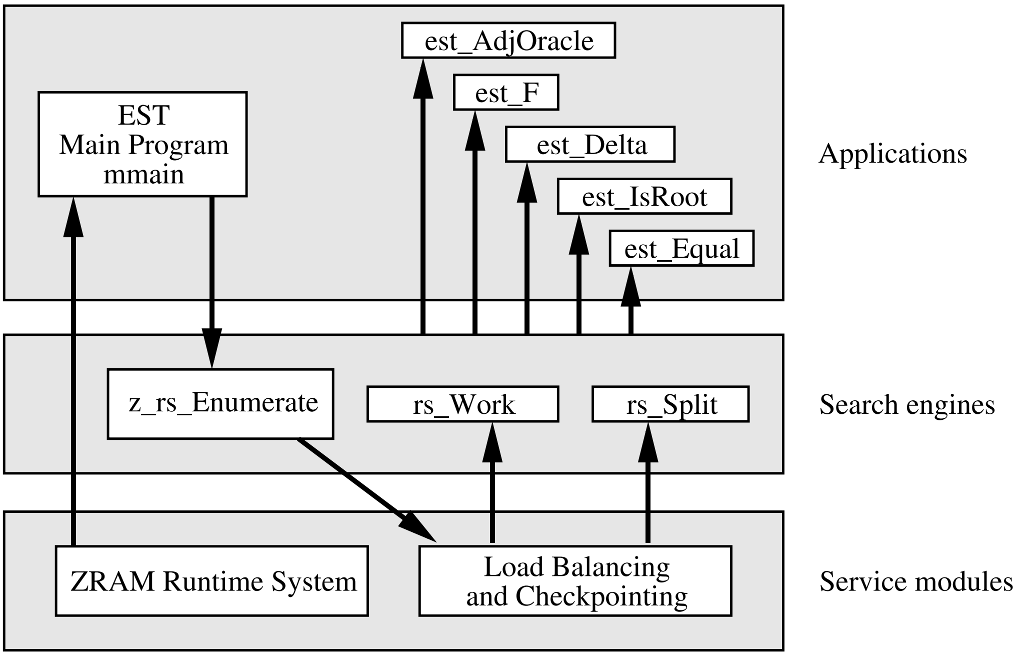

Figure 6.4: Mapping the reverse-search operations onto the virtual machine. The search-engine layer defines the operations work and split in terms of the five application-specific functions.

The work is not split into contiguous subtrees, but rather into intervals of the tree's preorder 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. The virtual machine requires the search engine to provide a *Work* and a *Split* operation on intervals (Sec. 5.3.3). These operations are defined as follows:

- *Work* removes the start node of an interval, decreasing the size of the interval by one.

- *Split* splits 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 a common ancestor, and k is the number of the node following e as a neighbor of the common ancestor defined by the adjacency oracle (Figure 6.5). The whole search tree is represented by the triple $\langle \text{root}, 0, \infty \rangle$. As the storage size of a node is typically much greater than two integers, this implementation outperforms one which simply stores the start and the end node.

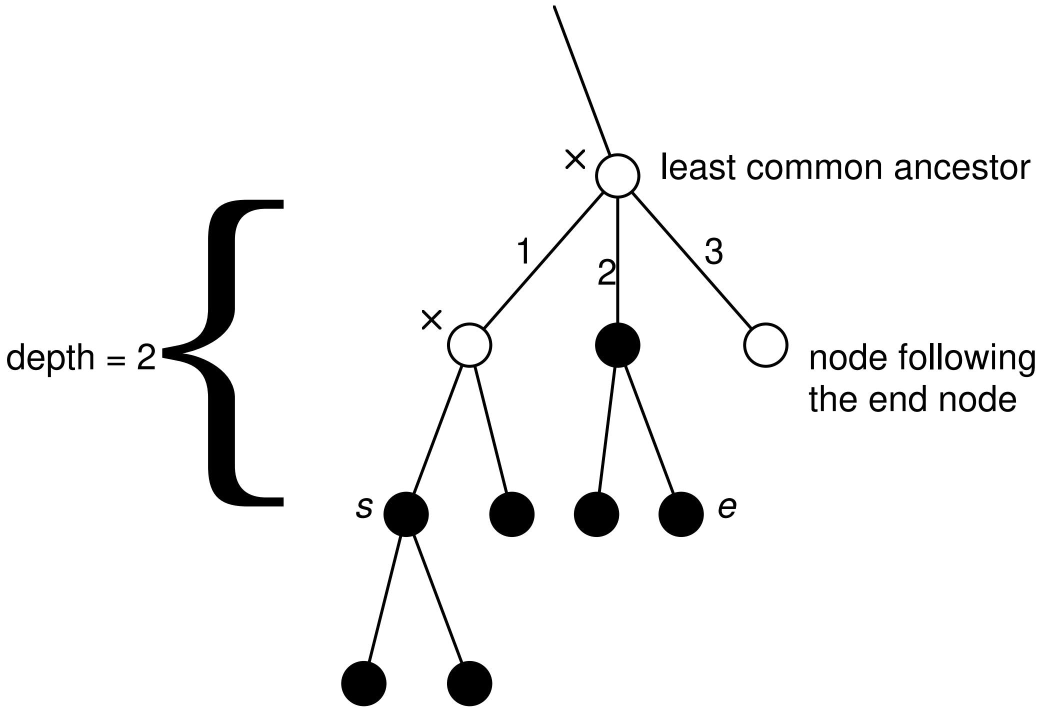

Figure 6.5: Mapping the reverse-search data onto the virtual machine. The filled circles are the elements of an interval of the preorder traversal represented by $\langle s, 2, 3 \rangle$. An interval corresponds to a work unit of the virtual machine. The nodes which may be in the ancestor cache are marked by a $\times$.

Discussion

The described implementation of the reverse-search engine has been chosen for its space-efficiency. This choice has the tradeoff that the split operation can be slow. It is the best choice when the memory size of the nodes is large. It is appropriate whenever the search tree is not too irregular and therefore few split operations are executed (for a speedup diagram, see Figure 7.1). But it can be improved when the nodes are so small that hundreds of them can be kept in memory. For these cases, a search engine similar to `z_bt_EnumerateDFExp` and `z_BranchBoundDFExp` should be designed. The work units given to the virtual machine would be subtrees rather than intervals.

6.4 Tree-Size Estimator

If we knew the running time of a given problem instance before starting the computation, we could avoid starting computations which later turn out to exceed the time available. This capability would be preferable to killing a computation which already has used many resources. We thus want to estimate the difficulty of a given instance so as to decide whether it is solvable by the available algorithm in a reasonable amount of time.

Because we cannot predict the running times of a Branch and Bound algorithm by traditional complexity analysis (aside from weak worst-case considerations), we need another approach for estimating the resources such as running time involved in the actual solution of an instance. Knuth's tree-size estimator [39] 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.

6.4.1 Interface

Application-Defined Functions

As the tree-size estimator is currently implemented only for branch-and-bound applications, it expects the same application-defined functions as the branch-and-bound engines, namely `Branch` or `BranchInPlace`, `Compare`, and `IsSolution` (see Section 6.2.1).

Search-Engine Calls

```
void z_BranchBoundEstimate(z_ProcSetNum procsetnum,
                           z_bb_GlobalDataP global,
                           z_NodeP root,
                           z_NodeP UpperBound,
                           int numEstimates,
                           float *nNodes,
                           float *sNodes,
                           float *nTime,
                           float *sTime)
```

This function estimates the size of a branch-and-bound search tree and the execution time of a sequential branch-and-bound algorithm. The parameters `procsetnum`, `global` and `root` are equivalent to the corresponding parameters of the branch-and-bound engine. `UpperBound` should be the value of an optimal solution, or an estimate of it, as the optimal solution usually is not available. The estimate of the tree-size depends heavily on the quality of `UpperBound`, for the estimator estimates the number of nodes which have a lower bound less than `UpperBound`. If `UpperBound` is too high, the tree size is overestimated; if it is too low, the tree size is underestimated.

`numEstimates` is the number of random paths through the tree to evaluate. The execution time of the estimator is proportional to it, and the variance of the estimates is inversely proportional to it. The four output parameters are described in Table 6.1.

Name	Meaning
`nNodes`	estimated number of nodes in the tree
`sNodes`	standard error of the estimated number of nodes
`nTime`	estimated execution time of the branch-and-bound engine
`sTime`	standard error of the estimated execution time

Table 6.1: Output of the tree-size estimator.

Discussion

The tree-size estimator could easily be extended to reverse-search and backtrack trees. This generalization has not been implemented yet because there was no demand for it.

6.4.2 Implementation

The parallel implementation of the tree-size estimator calculates the number of paths to be evaluated per processor (static load balancing), and broadcasts this number together with the global data and the root to the processors. The processors then independently evaluate the paths and collect the data needed. At the end of the computation, the front end gathers the results and computes the mean as well as the standard deviation as described in [39].

Discussion

The different paths in a search tree generally have different lengths and their evaluations have different running times, but the tree-size estimator, which has been parallelized before the load-balancer was available in the virtual machine, balances the load statically rather than dynamically. This implementation is no real handicap because usually thousands of paths are evaluated and because the execution time of every process depends only on the lengths of the chosen paths through the tree, which have a much smaller variance than the size of the subtrees.

Chapter 7

ZRAM in Action

Having explained the three lower layers of ZRAM, we will now illustrate its use for computation-intensive search problems. More than a dozen applications using ZRAM's search engines have been implemented. We start by describing the first parallel implementation of a space-efficient vertex enumeration algorithm for higher dimensions and a parallel solver for the quadratic assignment problem. These applications both have solved hitherto unsolved instances of difficult combinatorial problems. Both results involve large amounts of computation and would have been impossible without ZRAM's checkpointing possibility. We continue with four applications which show the flexibility of ZRAM, ending with references to other people who have used ZRAM profitably for the parallelization of search algorithms.

7.1 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 $\{x \in R^d : A\, x \leq b\}$, for some matrix $A \in R^{m \times d}$ and vector $b \in R^m$. The *vertex enumeration problem* consists of generating all the 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 by the duality of points and hyperplanes [3, 26], 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 [22].

The least time and space complexities for solving vertex enumeration and convex hull problems, at least under the assumption of nondegeneracy, are achieved by the reverse search algorithm [3], which is based on reversing the Simplex method for linear programming. The time and space complexities of the vertex enumeration problem for v vertices are $O(mdv)$ and $O(md)$, respectively. Reverse search is ideally suited for parallel computation, unlike other algorithms such as the double description method and its dual, the beneath-and-beyond method [22], which are memory-intensive sequential algorithms. Other advantages of reverse search are the small size and simple structure of its checkpoint files, and the possibility of estimating the output size without doing the full computation.

The implementation is based on David Avis' sequential lrs code, which implements the Simplex algorithm with lexicographic pivoting, and it uses the same format for input and output files. Note that each basis of the system $A\,x \leq b$ represents a vertex, but generally there are many bases representing the same vertex.

Table 7.1 lists the running times for a sample polytope on four different machines, and Figure 7.1 shows the speedup on the Paragon. The speedup of 35.2 for 100 processors is rather good, as the test problem is small compared to real instances. Higher-dimensional problems, with much wider and larger search trees, have a more favorable speedup. See Section 6.3.2 for an explanation of the parallelization overhead.

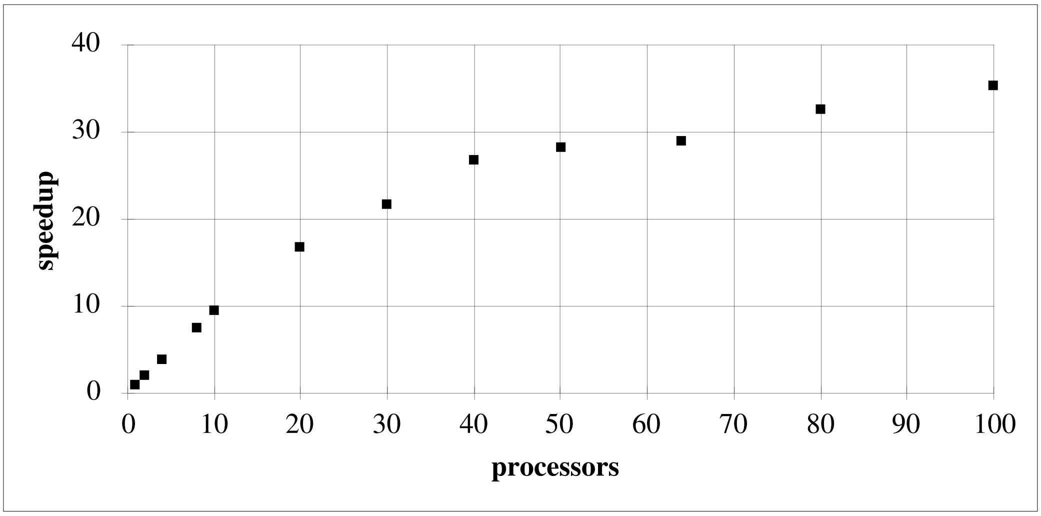

Figure 7.1: Speedup for vertex enumeration of C7-3 on the Intel Paragon.

Our parallel code was able to solve three large polytopes on a Cenju-3 with 64 processors. Table 7.2 shows their dimension, the number of inequalities

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

Table 7.1: Running times and speedup for the polytope c7-3 on different machines. 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 $v = 112$. The reverse search algorithm generates 75040 bases in a search tree of depth 37. Typical instances that arise in applications and have been solved by our ZRAM application are three orders of magnitude larger. As we can see, the Cenju-3 has the highest single-processor performance and (together with the Paragon) the highest speedup.

and vertices, the number of bases of the perturbed polytope, and the execution time. These instances could not be solved on any single workstation—the estimated CPU time on a DEC AXP workstation ranges from 130 days to 5.4 years. The Cenju-3 system we 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 while running reverse search was essential.

How can we be sure that the result of a month-long computation is correct? For vertex enumeration, one part of the verification is easy. Verifying that the points which are output are vertices of the polytope can be done in a reasonable amount of time. For the other part of the verification, "have all vertices been found?", no efficient algorithm is known.

Our work on this topic has led to the development of a primal–dual vertex-enumeration algorithm, which is described in [8, 7]. The new algorithm is based on the facts that (1) we can check the completeness of the vertices found by computing their convex hull and comparing it to the input, and (2) we can always derive an additional vertex from the comparison between an incomplete convex hull and the input.

Name	d	m	v	bases	time on Cenju-3 (64 procs)	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

Table 7.2: Three previously unsolved polytopes. The vertex-enumeration program based on ZRAM is suitable for computing polyhedra with billions of bases. The polytope MIT71-61 stems from materials science research and describes ground states of ternary alloys [16]. 01TORUS15X and 01TORUS16X describe reachable configurations in a variant of the well-known peg solitaire game [6, 2].

7.2 Quadratic Assignment Problem

The quadratic assignment problem (QAP) in Koopmans–Beckmann form can be stated as

$$\min_{\pi \in S} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i)\pi(j)}$$

where S is the set of all permutations of $1, 2, \ldots, n$, and F and D are integer $n \times n$ matrices. A typical example of a QAP is the facility location problem, in which a set of n facilities is to be assigned to an equal number of locations. Between each pair of facilities, there is a given amount of flow, contributing a cost equal to the product of the flow and the distance between the locations to which the facilities are assigned. There are many other applications [14, 51].

The QAP is a hard problem because many classical combinatorial optimization problems, such as the traveling salesman problem and the maximum clique problem, are special cases of the QAP. The exact solution of instances with $n \geq 20$ is reputed to be difficult, the only practical method for their solution being branch-and-bound. Although many sophisticated lower bounds for the QAP have been proposed, the most successful programs for solving the QAP use the rather weak bound developed by Gilmore and Lawler, which can be computed quickly.

In striking contrast to the difficulty of solving a QAP to optimality is the easyness of guessing optimal solutions by heuristic algorithms. A comparison of the main heuristics for the QAP can be found in [10].

We have implemented a parallel program [11, 12], which is based on

ZRAM, the Gilmore–Lawler bound, the branching rule of Mautor and Roucairol, and simulated annealing. The program first finds a good solution by simulated annealing, estimates the size of the search tree using ZRAM's estimator, and then verifies the solution by parallel depth-first branch and bound. As the initial solution is optimal in most cases, the depth-first search tree has the same size as a best-first tree would have. The program has little communication overhead and gives good speedup.

We used instances in the set of QAP instances that is commonly used to compare work on QAP algorithms (QAPLIB [15]) as a benchmark to evaluate our implementation. First we used the estimator on every unsolved instance to predict the execution time necessary for solving it. Then, we determined nine instances to be solvable within a reasonable amount of time and solved them. In this application, the tree-size estimator proved to be an extremely valuable tool. Instances HAD20 and TAI20A were solved on the Intel Paragon, the other instances on the NEC Cenju-3. Table 7.3 shows the execution time and the size of the search tree for all nine instances. The largest instance we could solve with this approach is NUG22 with almost 49 billion nodes in the search tree, and it was solved in 12 days with a varying number of processors.

name	cost	processors	nodes	time [Min]
HAD16	3 720	32	18 770 885	4
HAD18	5 358	16	761 452 218	442
HAD20	6 922	96	7 616 968 110	2875
TAI17A	491 812	32	20 863 039	6
TAI20A	703 482	96	2 215 221 637	684
ROU20	725 522	32	2 161 665 137	961
NUG21	2 438	16	3 631 929 368	3213
NUG22	3 596	48–96	48 538 844 413	12780
ESC32E	2	32	12 515 753	10

Table 7.3: Nine previously unsolved benchmark instances taken from QAPLIB. The tree-size estimator enabled us to select those instances that could be solved in the available amount of time.

After we had published our results, Tschöke et al. implemented a similar algorithm on their parallel PowerPC computer and solved the NUG24 instance. Their result motivated us to develop a dynamic programming algorithm based on a new relaxation of the QAP which is stronger than the Gilmore–Lawler bound. By running this algorithm on the Paragon, we could solve NUG25 [10].

7.3 Vertex Cover

We will now discuss an optimization problem arising in graph theory. Let $G = (V, E)$ be an undirected graph. A subset $U \subseteq V$ is called *vertex cover* if every edge of G has at least one endpoint in U (Figure 7.2); that is,

$$\forall (u, v) \in E : u \in U \vee v \in U$$

In the *vertex cover problem* we want to find a vertex cover with the smallest cardinality. This problem is equivalent to finding a maximum independent set in the same graph or finding a maximum clique in its complement. It is NP-complete [27], even for graphs containing no triangles. The vertex cover problem can be generalized to the *weighted vertex cover,* where every vertex has a weight and we want to find a cover of minimum weight.

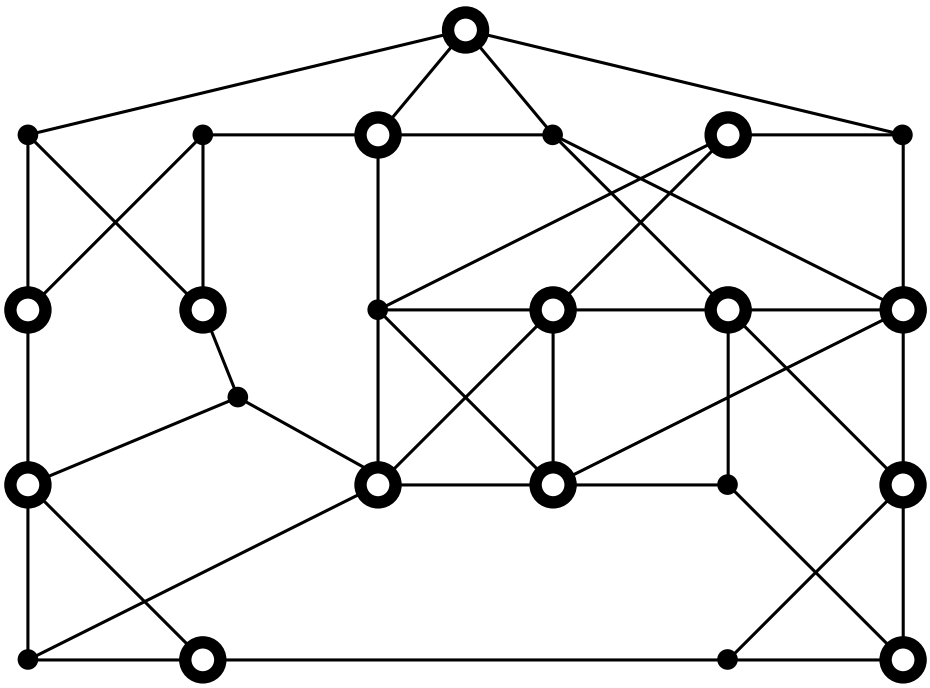

Figure 7.2: A minimal vertex cover of a graph. That the vertices marked with a ring cover all edges can easily be verified by hand: Check that all neighbors of every unmarked vertex are marked. Checking the minimality of the cover is an NP-complete problem.

Another generalization of vertex cover is the *set cover* problem. Given is a ground set $M = \{1, \ldots, m\}$ and set of n subsets $M_j \subseteq M$ with cost c_j, find a set $J \subseteq \{1, \ldots, n\}$ such that $\bigcup_{j \in J} M_j = M$ and $\sum_{j \in J} c_j$ is minimum. The definition of a_{ij} as 1 if $i \in M_j$ and 0 otherwise leads to the formulation as a zero-one optimization problem:

$$\begin{array}{lll} \text{Minimize} & \sum\limits_{j=1}^{n} c_j x_j & \\ \text{such that} & \sum\limits_{j=1}^{n} a_{ij} x_j \geq 1 & \text{for all } i \in \{1, \ldots, m\} \\ & x_j \in \{0,1\} & \text{for all } j \in \{1, \ldots, n\} \end{array}$$

ZRAM uses a branch-and-bound algorithm to find a minimum vertex cover for a graph. Although a branch-and-cut program might be more efficient [64], nobody has implemented branch-and-cut in parallel up to now, and branch-and-bound vertex-cover programs have been used by other authors [44] to test parallel algorithms. The unweighted vertex cover problem is an easy task for the dynamic load balancing as most nodes of the search tree have the same bound.

To optimize the performance for sparse graphs, the program uses an adjacency list of the graph as the main data structure. For dense graphs, an adjacency matrix would be more appropriate. The adjacency list is part of the global data and is not modified by the branching code.

The branching rule finds a vertex of maximum degree and generates two subproblems. Either the vertex is an element of the cover or it is not. If it is, all edges incident to it are covered and can be ignored. If it is not part of the cover, all vertices adjacent to it must be part of the cover and the edges incident to them can be ignored. The program simplifies the two subproblems by repeatedly marking and removing every vertex adjacent to a vertex of degree 1.

The data structure describing a subproblem (`vc_Node`) contains mainly an array of integers where the degree of every vertex is stored together with a flag signifying whether the vertex is part of the cover. This implementation makes the operations *find a maximum-degree vertex* and *mark a vertex and remove its adjacent edges* efficient. On the other hand, storing the redundant degree of every vertex wastes space. This redundancy is irrelevant for depth-first computations, but the range of instances that can be solved by best-first search is limited by the number of nodes that fit in the in-memory priority queue.

The most important step in determining a lower bound for a subproblem is the computation of a maximum matching by Edmonds' algorithm. The cardinality of the maximum matching is a lower bound for the vertex cover problem. This bound is never higher than 50 percent of the number of vertices in the graph. To achieve stronger bounds, we search for cliques of size ≥ 3

and odd-length circles. In a clique of size k, at least $k-1$ points are part of any cover. In a circle of length $2k-1$, at least k points are part of any cover. Of course, finding big cliques or many nonintersecting circles is a hard problem, so our algorithm uses a heuristic procedure to find a few of them quickly. All these ideas are applications of one principle: If any vertex cover for a graph is restricted to an induced subgraph, it covers this subgraph. Thus, if the vertices of a graph are partitioned into disjoint subsets, the sum of the sizes of minimum vertex covers for the subgraphs is a lower bound for the vertex cover of the original graph (Figure 7.3).

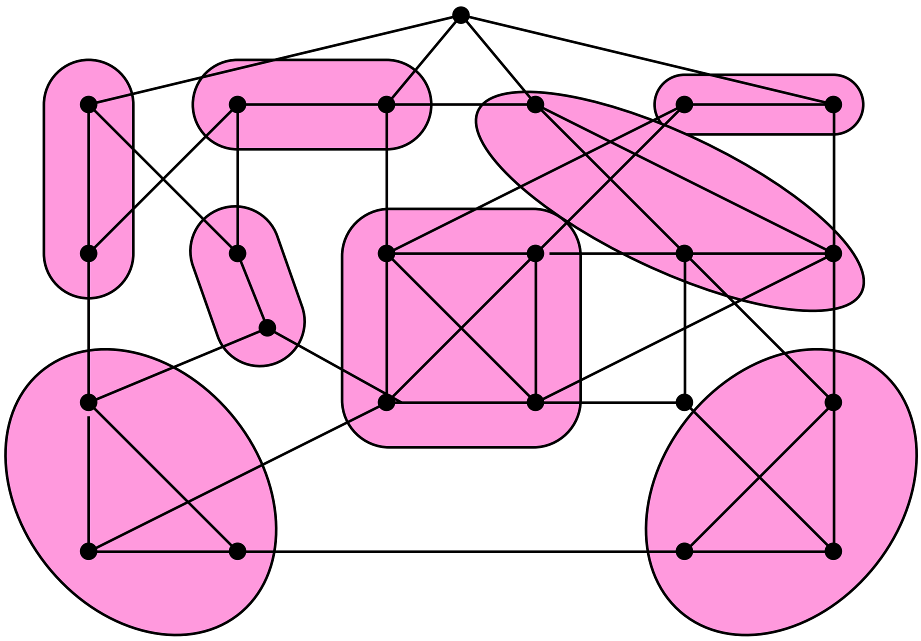

Figure 7.3: A lower bound for the minimum vertex cover. The graph is partitioned into subgraphs for which a minimum vertex cover can be computed easily. Every clique of size k has at least $k-1$ vertices in the cover, and thus $3+2+2+2+1+1+1+1=13$ is a lower bound for the size of a cover of the full graph.

7.4 Connected Induced Subgraphs

The enumeration of connected induced subgraphs (CIS) [4] is a typical application of the reverse-search paradigm. Given is an undirected graph $G=(V,E)$ with vertex set $V=\{1,2,\ldots,n\}$. We want to enumerate all connected induced subgraphs of G without incurring the overhead of enumerating all $2^{|V|}$ subsets of V.

In a reverse-search algorithm for CIS, the nodes of the search tree correspond to the connected subsets $U \subseteq V$. If $j \in U$ is the smallest vertex in U such that the subgraph induced by $U - j$ is connected, the local search function is defined as $f_{CIS}(U) := U - j$. This definition implies that the empty subgraph is the root of the search tree. The definition of the adjacency oracle can be deduced from our description of the local search function. The shallowness of the search tree (whose height is at most $|V|$) implies that it is wide, and thus makes good speedup possible. The CIS algorithm can also be used for the enumeration of polyominoes (Sec. 7.5).

The current implementation of the CIS algorithm on top of ZRAM is capable of producing text (a list of vertex sets) or graphical (PostScript, see Fig. 7.4) output. Its global data (`cis_Global`) consists of the number of vertices $|V|$, the adjacency lists, and the coordinates of the vertices if the output is to be shown graphically. A node of the search tree (`cis_Node`) stores the subset $U \subseteq V$.

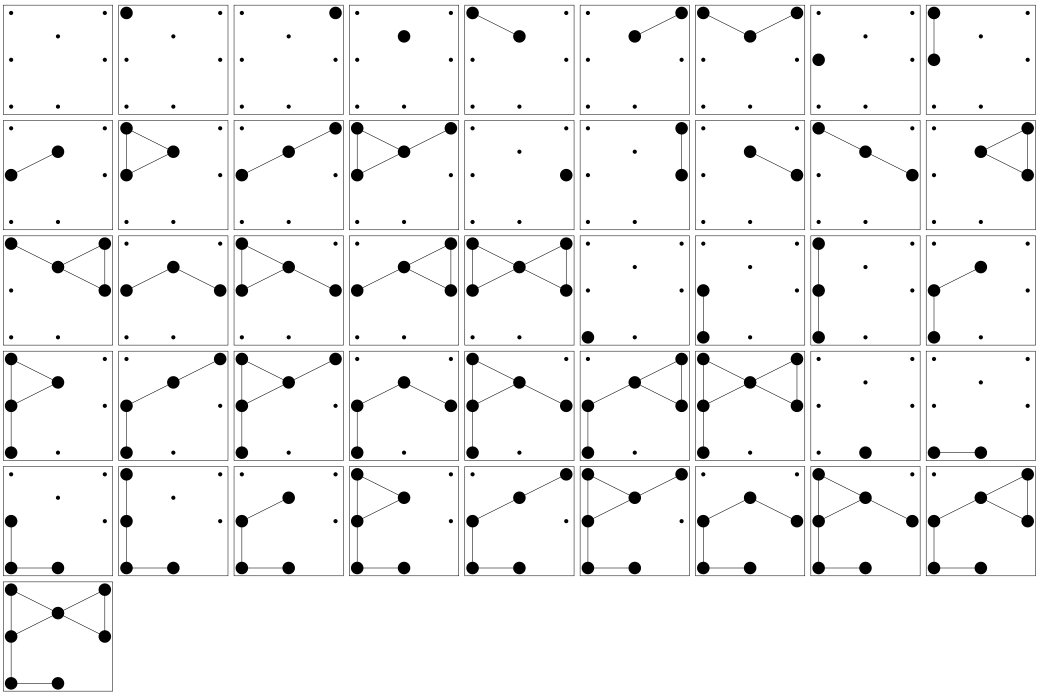

Figure 7.4: Connected induced subgraphs of a seven-vertex graph. The empty graph at the top left is the root of the reverse-search tree.

7.5 Polyominoes

Polyominoes are shapes made by connecting certain numbers of equal-sized squares, each joined together with at least one other square along an edge. Fig. 7.5 shows the 12 pentominoes (polyominoes consisting of 5 squares). The idea of polyominoes is old, but they were given their name only in 1953 by Golomb [32].

There is an exponential number of polyominoes; that is, there is a constant K (Klarner's constant) such that the number of polyominoes of size n is $\theta(K^n)$. The exact value of K is unknown, but has been proved to be between 3.9 and 4.649551.

Two different algorithms for the enumeration of polyominoes have been implemented on top of ZRAM. One of them (PMBT) uses backtracking, the other one (PMRS) uses the connected induced subgraphs program (Sec. 7.4) and thus is a reverse search application. Both algorithms generate symmetric instances of polyominoes but output only one representative of each class, and both of them produce graphical output (Figure 7.5 and Figure 7.6).

The reverse-search program models a polyomino as a connected induced subgraph of a suitably chosen rectangular-grid graph (Fig. 7.7). In fact, a smaller graph could be used, but this would complicate the detection of symmetries. The program enumerates all connected induced subgraphs of size 1 through n and prints those with size equal to n. It has a high overhead because of the implementation of the grid by adjacency lists, and is thus four times slower than the backtrack program (Fig. 7.8).

The backtrack program works on a rectangular array of tiles. Every tile is in one of the states *undecided, used,* or *free.* Initially one tile is *used,* and the rest is *undecided.* The algorithm recursively chooses an *undecided* tile adjacent to a *used* one and sets it either to *used* or to *free* until the polyomino has size n.

7.6 Euclidean Spanning Trees

Let P be a finite set of points in the plane, no three of which are collinear. We consider trees with vertices in P and edges given by line segments with endpoints in P, and we want to enumerate all of them that do not have intersecting edges. These trees are called Euclidean Spanning Trees (EST), and can be enumerated efficiently by reverse search [4].

The current implementation of the EST algorithm on top of ZRAM is capable of producing text (a list of trees represented as lists of edges) or graphical (PostScript) output (Figure 7.9). Its global data (`est_Global`)

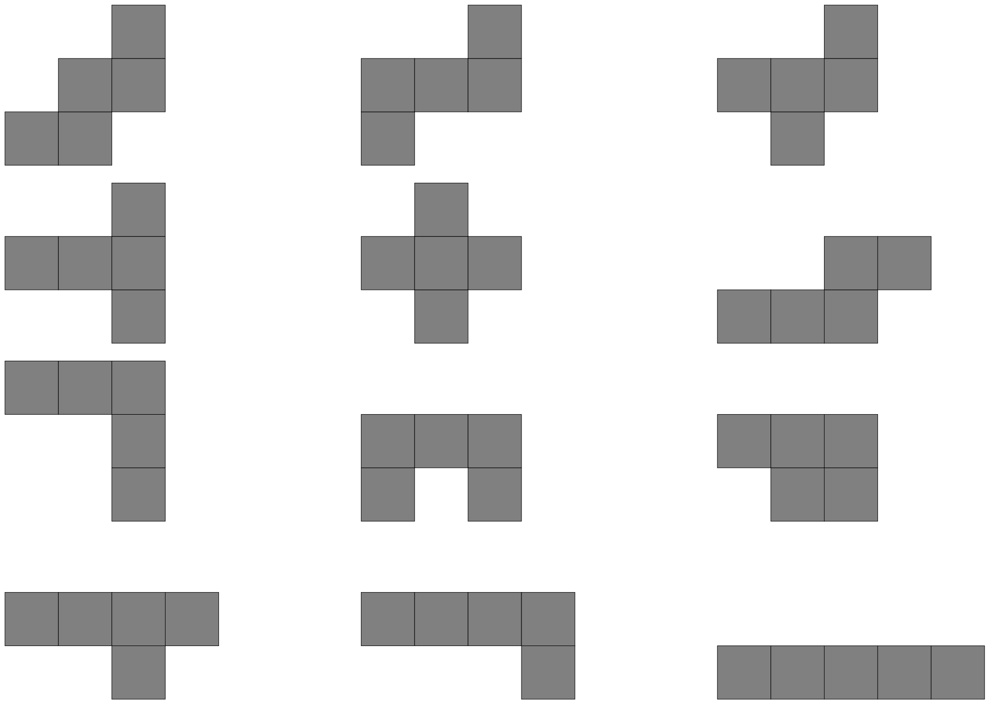

Figure 7.5: The 12 pentominoes.

consists of the number of points and their coordinates. A node of the search tree (`est_Node`) corresponds to a Euclidean Spanning Tree and is represented as an array storing the lexicographically sorted edges.

7.7 Other Applications

The following list of other ZRAM applications shows the flexibility of ZRAM and demonstrates that several people have learned to use the library productively within a few days.

- The n-queens problem was presented in Chapter 2.
- Another simple example of a backtrack algorithm is the enumeration of all partitions of a set into disjoint subsets. Instead of describing the algorithm, we show its output for a four-element set (Table 7.4).
- Brüngger [13, 10] implemented two versions of a 15-puzzle solver using branch-and-bound and Manhattan distances as lower bound (Figure 7.10). He used ZRAM for one version and carefully optimized the

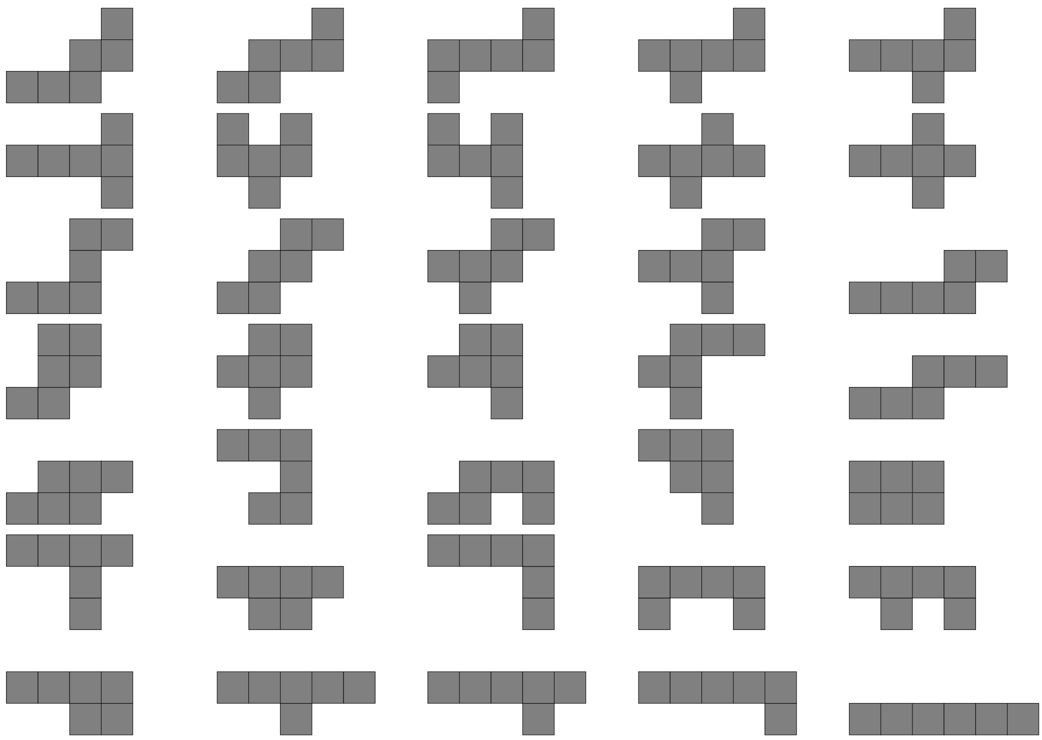

Figure 7.6: The 35 hexominoes. They can be enumerated by reverse search (based on the enumeration of connected induced subgraphs) and by backtracking.

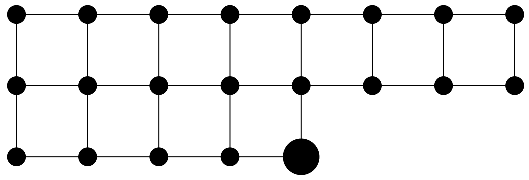

Figure 7.7: Mapping of polyominoes to connected induced subgraphs. In the reverse-search program, the pentominoes correspond to the connected induced subgraphs of the grid shown which have five vertices and contain the marked vertex.

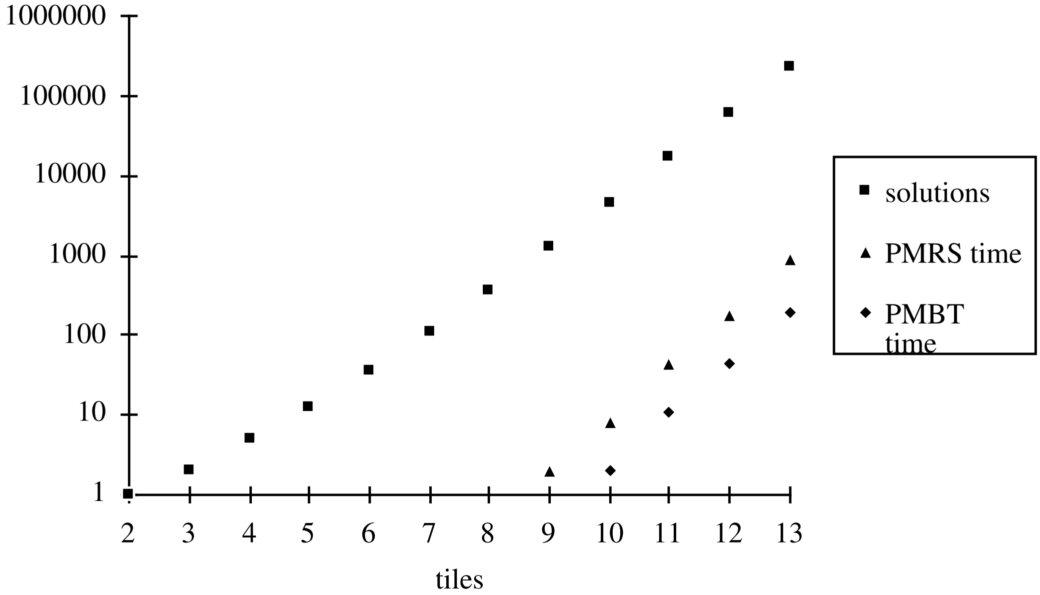

Figure 7.8: Number of polyominoes and sequential execution time of both implemented programs in seconds. The backtrack program is four times faster than the one built on top of the connected induced subgraphs application.

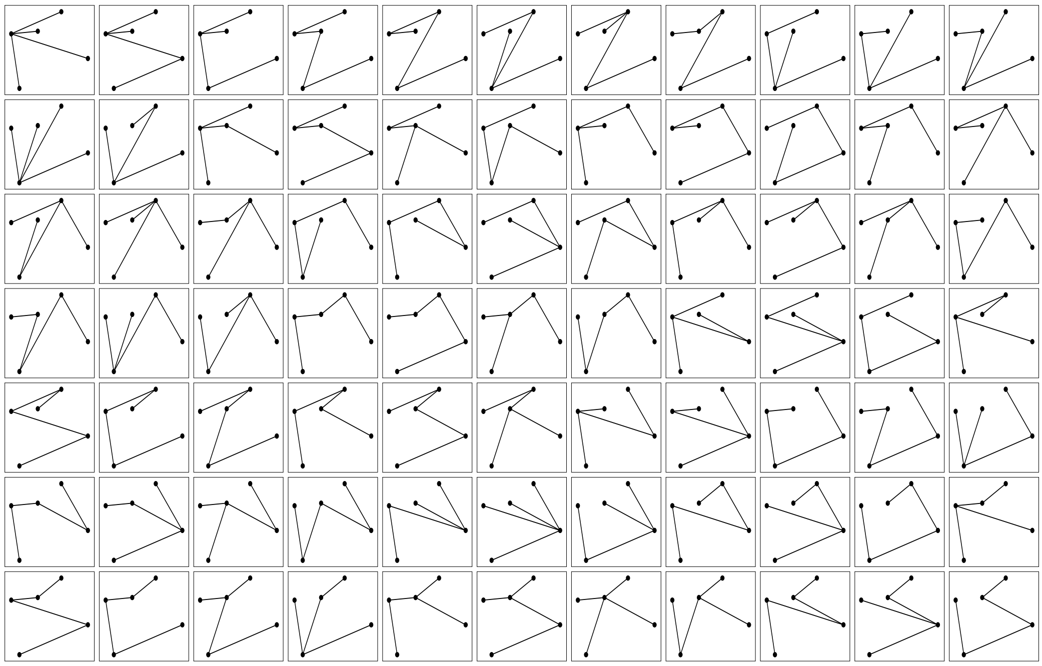

Figure 7.9: The 77 Euclidean spanning trees of a five-point set. The top-left tree is the root of the reverse-search tree.

$\{1,2,3,4\}$	$\{1,4\},\{2,3\}$
$\{1,2,3\},\{4\}$	$\{1,4\},\{2\},\{3\}$
$\{1,2,4\},\{3\}$	$\{1\},\{2,3,4\}$
$\{1,2\},\{3,4\}$	$\{1\},\{2,3\},\{4\}$
$\{1,2\},\{3\},\{4\}$	$\{1\},\{2,4\},\{3\}$
$\{1,3,4\},\{2\}$	$\{1\},\{2\},\{3,4\}$
$\{1,3\},\{2,4\}$	$\{1\},\{2\},\{3\},\{4\}$
$\{1,3\},\{2\},\{4\}$	

Table 7.4: Partitions of a four-element set. The 15 ways of partitioning the set $\{1,2,3,4\}$ into disjoint subsets are enumerated by a backtrack algorithm.

other one for speed without using a search library. He reports that the library version runs on a single processor at one-half of the speed of the optimized version. At first sight, the overhead incurred by using the library may seem to be large, but the 15-puzzle is a worst-case example because the processor can generate the moves and update the bounds in almost no time. In the QAP application, for instance, the overhead is negligible and cannot even be measured.

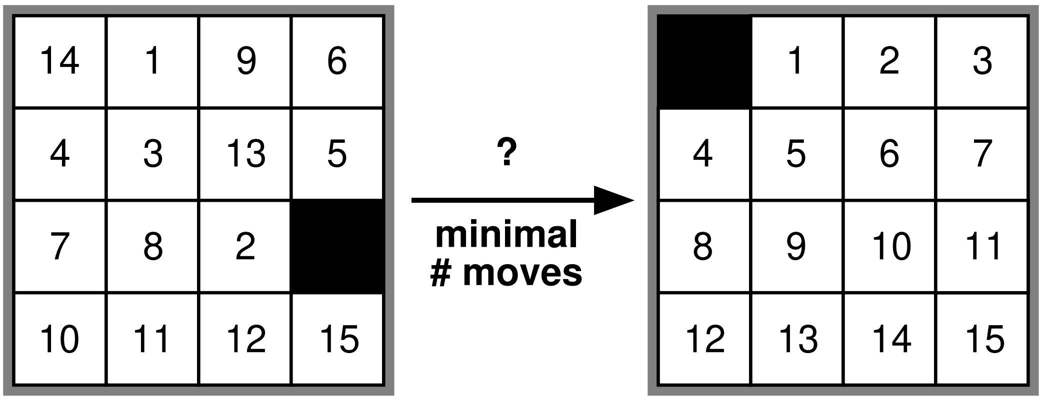

Figure 7.10: The 15-puzzle. Given any starting permutation of the 15 tiles, the problem is finding a minimal sequence of moves to the goal position. Brüngger [10] has proved that 80 moves always suffice and are necessary in the worst case. Figure courtesy of Adrian Brüngger, ETH Zürich.

- Brüngger [9] implemented a traveling salesman solver using the Held–Karp 1-Tree with iteration of Lagrangean coefficients as the lower bound.

- Sleumer [58] used ZRAM to parallelize two algorithms (reverse search and backtrack) that enumerate the cells in a hyperplane arrangement, a well-known geometric problem (Figure 7.11). For the backtrack algorithm, she determined isoefficiency curves (Figure 7.12).

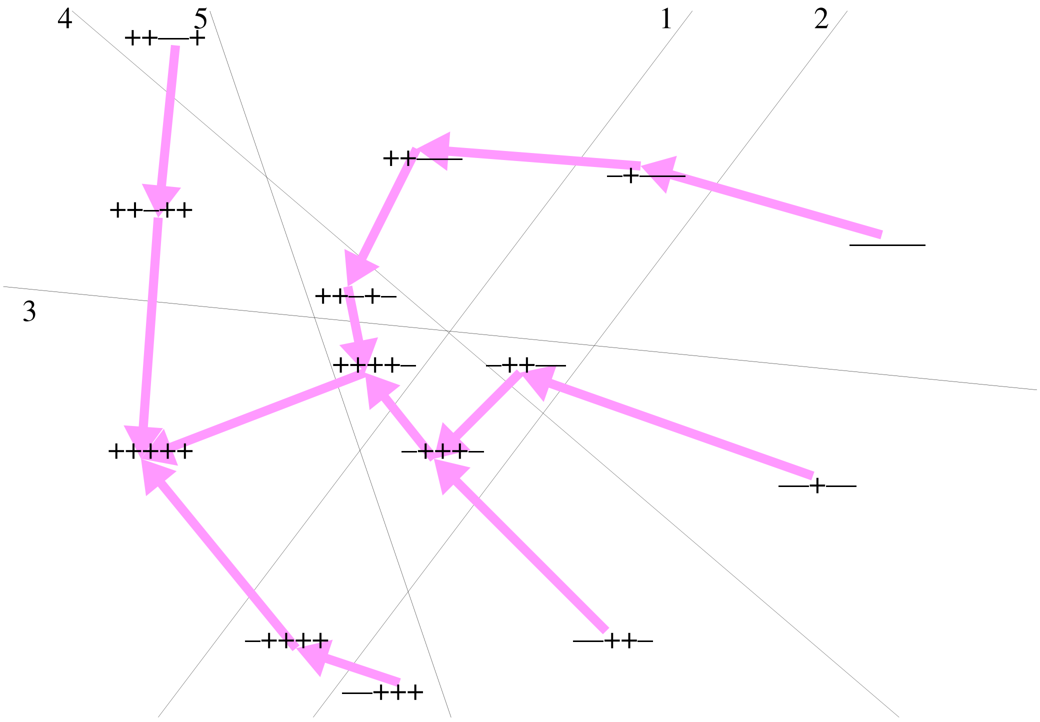

Figure 7.11: The cells of a two-dimensional arrangement. A reverse-search algorithm [4] enumerates the cells of an arrangement of m hyperplanes. Every cell is represented by its sign vector in $\{+,-\}^m$. The local search (parent) function is defined as the lexicographically largest neighbor of a cell, and the $+++++$-cell is the root of the search tree.

- Kindervater ported a branch-and-bound program for a scheduling problem to ZRAM. In a single day, he learnt to use ZRAM and ported his code.

- Fukuda implemented a branch-and-bound program for the knapsack problem and a reverse-search program for the enumeration of topological sortings.

- Sosnowska and Rolim used ZRAM's branch-and-bound engine for developing a parallel program for the fleet assignment problem.

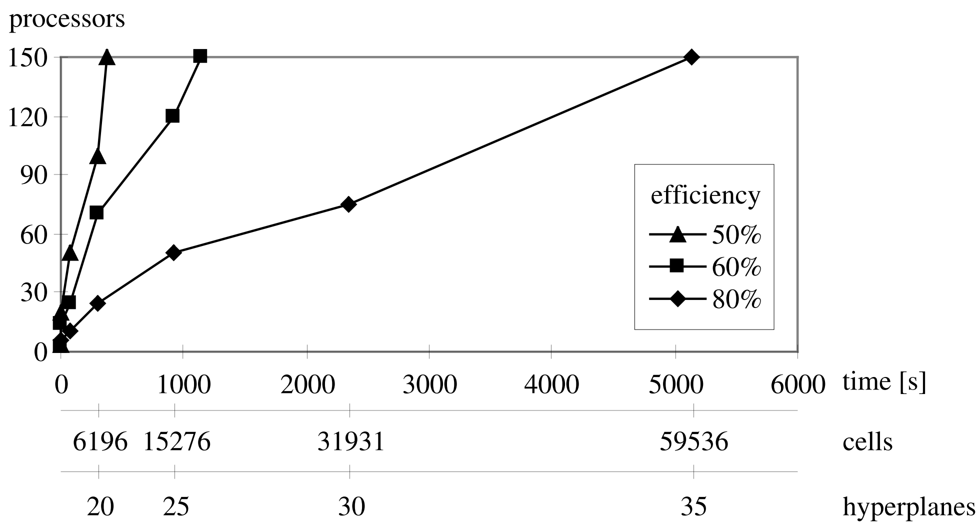

Figure 7.12: Isoefficiency for cell enumeration using a backtrack algorithm. We see contour lines of efficiency as a function of problem size (execution time of the sequential algorithm) and number of processors. The measurements were made on an Intel Paragon with random nondegenerate hyperplane arrangements in four dimensions, where m hyperplanes partition the space into $(m^4 - 2m^3 + 11m^2 + 14m + 24)/24$ cells. Figure courtesy of Nora Sleumer, ETH Zürich.

- In his diploma thesis, Ammeter developed a parallel program for the approximate solution of the vehicle routing problem using a GRASP algorithm. Instead of using a search engine, this program directly accesses ZRAM's virtual machine to balance the load dynamically.

Although several of the people mentioned had no experience with distributed computing at all when they began using ZRAM, they all quickly developed efficient applications based on its parallel search engines. How could there be a stronger evidence of ZRAM's flexibility and usability?

Chapter 8

Conclusions

With ZRAM, we have designed and implemented a software library which has been useful in the development of a wide range of applications. Some of its users had little or no experience in parallel programming and got access to parallel computers only through ZRAM.

ZRAM is the first search library containing a tree-size estimation tool, which proved to be valuable in allocating the limited CPU resources to the most promising problem instances. It contains the first parallel implementation of the reverse-search algorithm and the first parallel branch-and-bound engine which can be restarted at checkpoints.

The combination of these elements, together with the efficiency of the implementation, allowed us to solve large QAP instances and to enumerate the vertices of complex high-dimensional polytopes. These results would have been unachievable on a sequential workstation in a reasonable amount of time.

The work on ZRAM has clarified the requirements to a search library. The comparison of ZRAM to similar systems (e.g., BOB or PPBB) demonstrates that a useful parallel search library comprises much more than a collection of load-balancing algorithms or a branch-and-bound engine; it contains search engines for various paradigms, a tree-size estimator and a checkpointing facility. Further, we have seen that a four-layer structure is appropriate.

8.1 Lessons Learned

In the area of parallel search, libraries are a useful tool for several reasons: There is a wealth of applications which build on the same or similar search engines and whose common code is complex enough that learning to use a library is more economical than reimplementing and debugging the code for every application. The advantages of reusing code and of the short implementation times outweigh the few disadvantages. Although every library introduces a little overhead in execution time, the cost incurred is small compared to the work done in the application layer such as the computation of lower bounds or pivot operations. Libraries bring the power of parallel computers in an efficient way to people who are used to sequential machines and have little knowledge in parallel programming. Programs using a library are typically more clearly structured than those which do not. This effect again enhances the chances that program parts can be reused.

For large problem instances, the importance of a fine-tuned (and perhaps application-specific) implementation of a sophisticated load-balancing algorithm has been overestimated. The quality of load balancing affects only the very last minutes of a computation which takes hours. The coarse-grained parallelism of ZRAM generates little communication.

Finally, an advice to future library designers: Start by studying many potential applications of your planned library and implement some of them. Building on this experience, define the scope of the whole library (what applications and what host systems will be supported, etc.), and then specify all interfaces at once. The result of this process will be a library that is more clearly structured and easier to maintain than one which has grown during years. Some elements of ZRAM can only be explained historically and should be redesigned in a later version. For instance, the various depth-first search engines contain similar code which should exist only once and be shared by all of them.

8.2 Directions for Future Research

Research on parallel search libraries can and should proceed in numerous ways. Most obvious are the possible research directions on the search-engine level:

- Besides those implemented in ZRAM, there are other CPU-intensive search algorithms where an application-independent part can be separated from the application-dependent parts and be integrated into the library. Candidates are exact algorithms such as branch-and-cut, alpha–beta search and the evaluation of and–or trees, and approximation algorithms such as GRASP and simulated annealing. Some applications could profit from a general transposition table provided by the library or from other mechanisms to share data among branches of the search tree. Possible structures for the state space range from trees to directed acyclic graphs and general graphs.

- Currently, there exist half a dozen parallel search libraries including ZRAM and Ralph Gasser's SearchBench. Users would profit from a unified application interface for all of them.

- Parallel programming environments often have a graphical user interface, which shows data useful for performance tuning. How can we visualize the statistics which are typical for search algorithms (e.g., branching factors, lower bounds, or the number of cutoffs depending on the depth) such that the programmer most easily recognizes the essential information?

Extending the virtual machine and the host-system layer so that ZRAM (or another parallel search library) runs on more machines—possibly even on heterogeneous systems—raises the following issues:

- Porting the library to other host systems will make it usable for more people. Most distributed-memory implementations will be a matter of a few days. For an efficient shared-memory implementation, however, changes in the virtual machine and modified search algorithms might be necessary.

- On unreliable workstation networks, a fault-tolerant implementation of the load-balancing and checkpointing modules renders the computing power of underutilized workstations useful for long computations.

- As computers become cheaper from year to year, machines consisting of thousands of processors will soon become much more common than today. The algorithms in ZRAM have been used on up to 150 processors. Their scalability to machines which are ten times larger has not yet been investigated.

Finally, we see three open questions which are more theoretically oriented:

- There are techniques for converting parallel algorithms to external-memory algorithms. Would a best-first branch-and-bound storing its priority queue in, say, a Gigabyte of external memory be practical? Would it solve significantly larger problems than the usual internal-memory versions?

- The longer a computation is, the fewer people trust its correctness. What application-independent methods are there to detect hardware faults, software faults, and manipulation errors? Can ideas such as certification trails [63] or N-version programming be incorporated into a search algorithm library?

- In some optimization problems (e.g., the QAP), an optimal solution can easily be guessed, but the proof of its optimality takes much more time. What data must we show sceptics to convince them that the computation for the optimality proof really has been executed?

Bibliography

[1] Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization.* Wiley, 1997.

[2] David Avis and Antoine Deza. Solitaire cones. Technical Report 130, École des Hautes Études en Sciences Sociales, Centre d'Analyse et de Mathématique Sociales, 1996.

[3] David Avis and Komei Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Computational Geometry*, 8:295–313, 1992.

[4] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65:21–46, 1996.

[5] M. Benaïchouche, Van-Dat Cung, Salah Dowaji, Bertrand Le Cun, Thierry Mautor, and Catherine Roucairol. Building a parallel branch and bound library. In Ferreira and Pardalos [23], pages 201–231.

[6] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your mathematical plays 2: Games in Particular.* Academic Press, London, 1982.

[7] David Bremner, Komei Fukuda, and Ambros Marzetta. Primal–dual methods for vertex and facet enumeration. *Discrete and Computational Geometry.* to appear.

[8] David Bremner, Komei Fukuda, and Ambros Marzetta. Primal–dual methods for vertex and facet enumeration (extended abstract). In *ACM Symposium on Computational Geometry*, pages 49–56, 1997. `http://wwwjn.inf.ethz.ch/ambros/bfm97.ps.gz`.

[9] Adrian Brüngger. A parallel best-first branch and bound algorithm for the traveling salesperson problem. In S. Ranka, editor, *Proceedings of the 9th International Parallel Processing Symposium (IPPS'95), Workshop*

on *Solving Irregular Problems on Distributed Memory Machines*, pages 98–106, 1995.

[10] Adrian Brüngger. *Solving Hard Combinatorial Optimization Problems in Parallel: Two Case Studies.* PhD thesis, ETH Zürich, 1997.

[11] Adrian Brüngger, Ambros Marzetta, Jens Clausen, and Michael Perregaard. Joining forces in solving large-scale quadratic assignment problems in parallel. In IPPS'97 [38], pages 418–427. `http://wwwjn.inf.ethz.ch/ambros/ipps97_qap_zram.ps.gz`.

[12] Adrian Brüngger, 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.

[13] Adrian Brüngger, Ambros Marzetta, Komei Fukuda, and Jürg Nievergelt. The parallel search bench ZRAM and its applications. *Annals of Operations Research*. to appear; `http://wwwjn.inf.ethz.ch/ambros/aor_zram.ps.gz`.

[14] Rainer E. Burkard and Eranda Çela. Quadratic and three-dimensional assignments. In Dell'Amico et al. [20], chapter 21, pages 373–391.

[15] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. QAPLIB—a quadratic assignment problem library. *Journal of Global Optimization*, 10:391–403, 1997.

[16] 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 B*, 49(1):1–7, January 1994.

[17] Xinghao Chen and Michael L. Bushnell. *Efficient Branch and Bound Search with Application to Computer-Aided Design*. Kluwer Academic Publishers, 1996.

[18] R. Corrêa and A. Ferreira. Parallel best-first branch-and-bound in discrete optimization: a framework. In Ferreira and Pardalos [23], pages 171–200.

[19] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[20] Mauro Dell'Amico, Francesco Maffioli, and Silvano Martello, editors. *Annotated Bibliographies in Combinatorial Optimization.* John Wiley and Sons, 1997.

[21] R. Diekmann, R. Lüling, and J. Simon. A general purpose distributed implementation of simulated annealing. In SPDP'92 [59], pages 94–101.

[22] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer, 1987.

[23] Afonso Ferreira and Panos Pardalos, editors. *Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science.* Springer, 1996.

[24] Afonso Ferreira and José Rolim, editors. *Parallel Algorithms for Irregularly Structured Problems, IRREGULAR '95*, volume 980 of *Lecture Notes in Computer Science.* Springer, 1995.

[25] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9:235–256, 1987.

[26] Komei Fukuda and Alain Prodon. Double description method revisited. In M. Deza, R. Euler, and I. Manoussakis, editors, *Combinatorics and Computer Science*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer-Verlag, 1996. `ftp://ftp.ifor.math.ethz.ch/pub/fukuda/reports`.

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

[28] Ralph Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search.* PhD thesis, ETH Zürich, 1995.

[29] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[30] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[31] F. Glover, M. Laguna, E. Taillard, and D. de Werra, editors. *Tabu Search*, volume 41 of *Annals of Operations Research.* Baltzer, 1993.

[32] Solomon W. Golomb. *Polyominoes.* Princeton University Press, 1994.

[33] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[34] Charles Antony Richard Hoare. *Communicating sequential processes*. Prentice-Hall International, 1985.

[35] Kien A. Hua, Wen K. Lee, and S. D. Lang. On random sampling for parallel simulated annealing. In IPPS'94 [37], pages 253–257.

[36] *Proc. of the 6th International Parallel Processing Symposium, (IPPS '92)*. IEEE Computer Society Press, 1992.

[37] *Proceedings of the 8th International Parallel Processing Symposium*. IEEE Computer Society Press, 1994.

[38] *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, 1997.

[39] D. E. Knuth. Estimating the efficiency of backtrack programs. *Math. Comp.*, 29:121–136, 1975.

[40] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[41] Norbert Kuck, Martin Middendorf, and Hartmut Schmeck. Generic branch-and-bound on a network of transputers. In R. Grebe et al., editors, *Transputer Applications and Systems '93*, pages 521–535. IOS Press, 1993.

[42] G. Laporte and I.H.Osman, editors. *Metaheuristics in Combinatorial Optimization*, volume 63 of *Annals of Operations Research*. Baltzer, 1996.

[43] Shyh-Chang Lin, W. F. Punch III, and E. D. Goodman. Coarse-grain parallel genetic algorithms: Categorization and new approach. In SPDP'94 [60], pages 28–37.

[44] R. Lüling and B. Monien. Load balancing for distributed branch & bound algorithms. In IPPS'92 [36], pages 543–549.

[45] Friedemann Mattern. Experience with a new distributed termination detection algorithm. In Proceedings [50], pages 127–143.

[46] G. P. McKeown, V. J. Rayward-Smith, and H. J. Turpin. Branch-and-bound as a higher-order function. *Annals of Operations Research*, 33:379–402, 1991.

[47] David Nassimi, Milind Joshi, and Andrew Sohn. H-PBS: A hash-based scalable technique for parallel bidirectional search. In SPDP'95 [61], pages 414–419.

[48] Jakob Nielsen. The usability engineering life cycle. *Computer*, pages 12–22, March 1992.

[49] P. M. Pardalos, L. Pitsoulis, T. Mavridou, and M. G. C. Resende. Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and GRASP. In Ferreira and Rolim [24], pages 317–331.

[50] *Distributed Algorithms 1987*, volume 312 of *Lecture Notes in Computer Science*. Springer, 1987.

[51] *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1993.

[52] V.J. Rayward-Smith, I.H. Osman, C.R. Reeves, and G.D. Smith, editors. *Modern Heuristic Search Methods*. John Wiley and Sons, 1996.

[53] Roland Rühl. *A Parallelizing Compiler for Distributed Memory Parallel Processors*. PhD thesis, ETH Zürich, 1992.

[54] Sartaj Sahni and Venkat Thanvantri. Parallel computing: Performance metrics and models. Technical report, Florida University, 1996. `http://www0.cise.ufl.edu/research/tech-reports/tr96-abstracts.shtml`.

[55] Peter Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. PhD thesis, Karlsruhe, 1997.

[56] Yuji Shinano, Kenichi Harada, and Ryuichi Hirabayashi. Control schemes in a generalized utility for parallel branch-and-bound algorithms. In IPPS'97 [38], pages 621–627.

[57] Yuji Shinano, Masahiro Higaki, and Ryuichi Hirabayashi. A generalized utility for parallel branch and bound algorithms. In SPDP'95 [61], pages 392–401.

[58] Nora Sleumer. Output-sensitive cell enumeration in hyperplane arrangements. In *The Sixth Scandinavian Workshop on Algorithm Theory*, volume 1432 of *Lecture Notes in Computer Science*. Springer, 1998. to appear, `http://wwwjn.inf.ethz.ch/group/publications.html`.

[59] *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing 1992*. IEEE Computer Society Press, 1992.

[60] *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing 1994*. IEEE Computer Society Press, 1994.

[61] *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing 1995*. IEEE Computer Society Press, 1995.

[62] Lawrence D. Stone. *Theory of Optimal Search*, volume 118 of *Mathematics in Science and Engineering*. Academic Press, 1975.

[63] Gregory F. Sullivan, Dwight S. Wilson, and Gerald M. Masson. Certification of computational results. *IEEE Transactions on Computers*, 44:833–847, 1995. `http://www.cs.jhu.edu/~sullivan/download.html`.

[64] Stefan Thienel. *ABACUS: A Branch-And-CUt System*. PhD thesis, Universität Köln, 1995.

[65] Anestis A. Toptsis. Parallel bidirectional heuristic search with dynamic process re-direction. In IPPS'94 [37], pages 242–247.

[66] Stefan Tschöke and Norbert Holthöfer. A new parallel approach to the constrained two-dimensional cutting stock problem. In Ferreira and Rolim [24], pages 285–300.

[67] Stefan Tschöke and Thomas Polzer. Portable parallel branch-and-bound library: User manual. Technical report, University of Paderborn, 1995. `http://www.uni-paderborn.de/~ppbb-lib`.

Index

Curriculum Vitae

1986	A-Maturity, Humanistisches Gymnasium Basel.
1986	Software development at Contraves AG, Zürich.
1986–1991	Major in computer science and minor in electrical engineering at ETH Zürich, resulting in the degree of Dipl. Informatik-Ing. ETH.
1991	Development of an application-specific integrated circuit at Landis & Gyr Building Control AG, Zug.
1991–1998	Assistant and Ph.D. student in the research group of Prof. Nievergelt, Institute of Theoretical Computer Science, ETH Zürich.