Noname manuscript No.
(will be inserted by the editor)

A portable parallel implementation of the *lrs* vertex enumeration code

David Avis and Gary Roumanis

the date of receipt and acceptance should be inserted later

Abstract We describe a parallel implementation of the vertex enumeration code *lrs* that automatically exploits available hardware on multi-core computers and runs on a wide range of platforms. The implementation makes use of a C++ wrapper that essentially uses the existing *lrs* code with only minor modifications. This allows the simultaneous development of the existing single processor code with the speedups available from multi-core systems. It makes use of the restart feature of reverse search that allows for independent subtree search and the fact that no communication is required between these searches. As such it can be readily adapted for use in other reverse search enumeration codes.

Keywords vertex enumeration, reverse search, parallel processing

Mathematics Subject Classification (2000) 90C05

1 Introduction

Since its discovery in the 1990s the reverse search technique[3] [4] has been used to solve a large number of unstructured enumeration problems of which perhaps the most widely used is vertex enumeration using the *lrs* program [2]. From the outset it was realized that reverse search was imminently suitable for parallelization. The first such code, *prs* was developped by Ambros Marzetta using his ZRAM parallelization platform, as described in [6] and

David Avis
School of Informatics, Kyoto University, Kyoto, Japan and
School of Computer Science, McGill University, Montréal, Québec, Canada
E-mail: avis@cs.mcgill.ca

Gary Roumanis
Microsoft Research, Seattle, USA
E-mail: groumanis@gmail.com
This work is supported by NSERC and JSPS.

available online at [10]. In this case the parallelization was built into the *lrs* code itself leading to problems of maintennance and upgrading as newer parallel libraries developed. Another parallelization of reverse search was developed by Christophe Weibel for computing Minkowski sums [11]. This used a recursive version of reverse search where a backtrack stack is employed and some message passing is allowed during parallel execution. We discuss it further in Section 2.3.

The *lrs* code is rather complex and has been under development for over twenty years incorporating a multitude of different functions. It has been used extensively and basic functionality is very stable. Directly adding parallelization code to such legacy software is extremely delicate and can easily produce bugs that are difficult to find. The approach we use avoids this completely as the parallelization occurs in a separate layer. This allows independent development of both parallelization ideas and basic improvements in the underlying code. Parallelization is obtained by using the built in restart features of *lrs* with a completely separate multi-thread scheduler. The concept was tested by a shell script, *tlrs* developed by John White in 2009. Here the parallelization is achieved by scheduling independent processes for subtrees via the shell. Although good speedups were obtained several limitations of this approach materialized as the number of processors available increased. In particular job control becomes a major issue: there is no single controlling process. A strong point of the approach used in *tlrs* was that no modification of the underlying *lrs* code was required.

The approach we describe here lies somewhere between the approaches of Marzetta and White. We built a C++ wrapper that compiles in the original *lrslib* library essentially maintaining the integrity of the underlying *lrs* code. The parallelization is achieved by multithreading using an initial bounded depth run of *lrs* and and an additional process is used to concatenate the output streams. Job control is easily available since one process is in charge of all threads. Furthermore the development of the parallelization techniques can proceed independently of the original *lrs* code itself.

The paper is organized as follows. In the next section we begin with background on reverse search and explain the simple modifications necessary to prepare for parallelization. We use the example of generating permutations as an illustration. We then give a high level description of the parallelization technique illustrating on the permutation example. In Section 3 we describe the vertex enumeration problem and some of the properties that may potentially limit parallel speedups. In Section 4 we describe the wrapper constructed to schedule the parallel *lrs* executions, detailing various design decisions taken. Section 5 gives numerical experiments on a wide variety of polyhedra with bench marks against the standard solvers *cddr+* [8] and *lrs* . We conclude with some observations and directions for improving the parallelization performance.

2 Background

2.1 Reverse search

Reverse search is a technique for generating large, relatively unstructured, sets of discrete objects. We give an outline of the method here referring the reader to [3] [4] for further details.

In its most basic form, it can be viewed as the traversal of a spanning tree, called the reverse search tree T, of a graph $G = (V, E)$ whose nodes are the objects to be generated. Edges in the graph are specified by an adjacency oracle, and the subset of edges of the reverse search tree are determined by an auxiliary function, which can be thought of as a local search function f for an optimization problem defined on the set of objects to be generated. One vertex, v^*, is designated as the *target* vertex. For every other vertex $v \in V$ repeated application of f must generate a path in G from v to v^*. The set of these paths defines the reverse search tree T, which has root v^*.

A reverse search is initiated at v^*, and only edges of the reverse search tree are traversed. When a node is visited the corresponding object is output. Since there is no possibility of visiting a node by different paths, the nodes are not stored. Backtracking can be performed in the standard way using a stack, but this is not required as the local search function can be used for this purpose. This means that it is not necessary to keep more than one node of the tree at any given time, and this memoryless property is the main feature of reverse search. For a given problem, there may be many choices of adjacency oracle and local search function.

However, in the basic setting described here, a few properties are required. Firstly, the underlying graph G must be connected and an upper bound on the maximum vertex degree, Δ, must be known. The performance of the method depends on G having Δ as low as possible. The adjacency oracle must be capable of generating the adjacent vertices of some given vertex v sequentially and without repetition. This is done by specifying a function $Adj(v, j)$, where v is a vertex of G and $j = 1, 2, ..., \Delta$. Each value of $Adj(v, j)$ is either a vertex adjacent to v or null. Each vertex adjacent to v appears precisely once as j ranges over its possible values. For each vertex $v \neq v^*$ the local search function $f(v)$ returns the tuple (u, j) where $v = Adj(u, j)$ such that u is v's parent in T. The algorithm is shown in Algorithm 1. The order that the vertices are output is called the *reverse search order*. For convenience later, we do not output the root vertex v^*.

These ideas can be illustrated on a simple example: generating all permutations of a set of integers. Here the goal is to generate all pemutations of the integers $\{1, 2, ..., n\}$. The underlying graph $G_n = (V, E)$ is defined as follows. The vertices are n-tuples, $v = v_1 v_2 ... v_n$, representing the $n!$ permutations of the n integers. We set the target $v^* = (12...n)$. The adjacency oracle simply interchanges two consecutive integers in a permutation, and is given by

$$Adj(v, i) = (v_1 v_2 ... v_{i-1} v_{i+1} v_i ... v_n) \qquad i = 1, 2, ..., n-1.$$

Algorithm 1 Generic Reverse Search

```
1: procedure RS(v*, Δ, Adj, f)
2:     v ← v* j ← 0
3:     repeat
4:         while j < Δ do
5:             j ← j + 1
6:             if f(Adj(v, j)) = v then                    ▷ forward step
7:                 v ← Adj(v, j)
8:                 output v
9:                 j ← 0
10:            end if
11:        end while
12:        if v ≠ v* then                                  ▷ backtrack step
13:            (v, j) ← f(v)
14:        end if
15:    until v = v* and j = Δ
16: end procedure
```

So G_n is regular of degree $\Delta = n - 1$. Finally we set the local search function f to interchange the first two consecutive integers that are out of order numerically:

$$f(v) = (v_1 v_2 ... v_{i-1} v_{i+1} v_i ... v_n) \quad \text{for the smallest} \;\; i \; s.t. \; v_i > v_{i+1}.$$

Figure 1 shows G_4 which has 24=4! vertices and is regular of degree 3. The edges chosen by the local search function f are shown with arrows directed towards the root 1234. So for example starting at vertex 4231 f generates the path

$$4231 \mapsto 2431 \mapsto 2314 \mapsto 2134 \mapsto 1234.$$

The set of all arcs with arrows defines the reverse search tree T.

2.2 Extended reverse search

To achieve parallelization of Algorithm 1 we make use of the lack of memory property that allows it to be restarted from any node in the reverse search tree T. After a restart, all remaining nodes of T will be generated. We adapt this to allow for a subtree to be enumerated from its given root.

When calling the reverse search procedure we now supply four additional parameters:

- *start_vertex* is vertex from which the reverse search should be initiated and replaces v^*
- *depth* is initially the depth in T of *start_vertex* and will be updated to be the depth in T of the vertex v currently being considered in the search
- *max_depth* is the depth at which forward steps are terminated
- *min_depth* is the depth at which backtrack steps are terminated

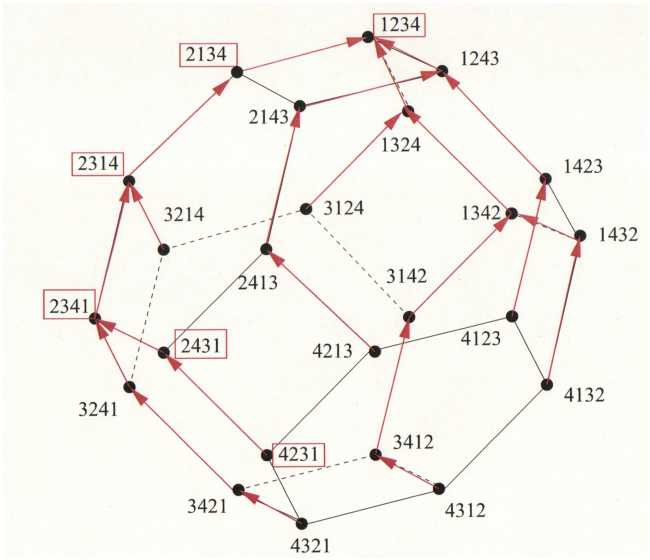

Fig. 1 Permutahedron: $n = 4$

Algorithm 2 Extended Reverse Search

```
1: procedure RS2(start_vertex, Δ, Adj, f, depth, max_depth, min_depth )
2:     j ← 0   v ← start_vertex
3:     repeat
4:         while j < Δ and depth < max_depth  do
5:             j ← j + 1
6:             if f(Adj(v, j)) = v then                          ▷ forward step
7:                 v ← Adj(v, j)
8:                 output v
9:                 j ← 0
10:                depth ← depth + 1
11:            end if
12:        end while
13:        if depth > 0 then                                     ▷ backtrack step
14:            (v, j) ← f(v)
15:            depth ← depth − 1
16:        end if
17:    until depth = min_depth and j = Δ
18: end procedure
```

The modified algorithm is shown in Algorithm 2.

Comparing Algorithm 1 and Algorithm 2 it is clear that the modifications are very simple. However they enable us to extend the function of Algorithm 1 in several ways. For any vertex v in T we denote its depth by $depth(v)$. Initially we have $depth(v^*) = 0$. For the generic version of reverse search we set $start\_vertex = v^*, depth = min\_depth = 0$ and $max\_depth = +\infty$. For a restart from vertex v we set $start\_vertex = v, depth = depth(v), min\_depth = 0$ and $max\_depth = +\infty$. To output all nodes in the subtree of T rooted at v we set $start\_vertex = v, depth = depth(v), min\_depth = depth(v)$ and

$max\_depth = +\infty$. To initialize the parallelization process we will generate the tree T down to a fixed depth k by setting $start\_vertex = v^*, depth = min\_depth = 0$ and $max\_depth = k$.

Returning to the example in Figure 1 we could do a restart from $v = 2143$ with $depth = 2$ obtaining the output 2413 4213 1423 4123. To list all nodes in the subtree rooted at v we would in addition set $min\_depth = 2$ producing the output 2413 4213. To do a partial enumeration down to $depth = 2$ we would set $start\_vertex = 1234, depth = min\_depth = 0, max\_depth = 2$ generating the output 2134 2314 1324 3124 1342 1243 2143 1423.

2.3 Parallelization

In this subsection we describe how the extended reverse search algorithm can be parallelized without requiring further modification. We give a rather generic description of the parallelization which is by nature somewhat oversimplified. The details of the actual implementation with the *lrs* program will be given in Section 4

We proceed in three phases. In the first phase we generate the reverse search tree T down to a fixed depth $init\_depth$. Rather than ouput the nodes of the tree, we store them in a list L. In the second phase we schedule threads in parallel from L using the subtree enumeration feature. For this we require the parameter $max\_threads$ giving the maximum number of parallel threads to that can run at the same time. We will also control where the output stream is sent. In Phase 1 it will be directed to the list L. From L all vertices that have depth less than $init\_depth$ are removed and output. In Phase 2 we schedule parallel threads from the nodes in L using the subtree enumeration feature. When the list L becomes empty we move to Phase 3 in which the parallel threads terminate one by one until there are no more running and the procedure terminates. We make use of a *collection process* which concatenates the output from the parallel threads into a single output stream. The procedure is outlined in Algorithm 3.

It is clear from the pseudocode the only interaction between the parallel threads is the common output collection process. The only signalling required is when a thread terminates.

Let us return to the example in Figure 1. Suppose we set the $init\_depth = 2$ and $max\_threads = 3$. We initiate the computation with the call

$$PRS(1234, 3, Adj, f, 2, 3)$$

This will generate the output list

$$L = \{2134\ 2314\ 1324\ 3124\ 1342\ 1243\ 2143\ 1423\}$$

in line 3. In line 6 we remove and output 2134 1324 1243 which have $depth < 2$ leaving $L = \{2314\ 3124\ 1342\ 2143\ 1423\}$ which are at $depth = 2$. We assume L is processed in left to right order. In lines 8-10 we initiate three calls to RS2:

$RS2(2314, 3, Adj, f, 2, \infty, 2), RS2(3124, 3, Adj, f, 2, \infty, 2),$ and$RS2(1342, 3, Adj, f, 2, \infty, 2).$

Algorithm 3 Parallel Reverse Search

```
 1: procedure PRS(start_vertex, Δ, Adj, f, init_depth, max_threads )
 2:     num_threads ← 0
 3:     redirect output to a list L                                  ▷ Phase 1
 4:     RS2(start_vertex, Δ, Adj, f, 0, init_depth, 0)
 5:     redirect output to collection process
 6:     remove all v ∈ L with depth(v) < init_depth and output(v)
 7:     while num_threads < max_threads and L ≠ ∅ do                 ▷ Phase 2
 8:         remove any v ∈ L
 9:         RS2(v, Δ, Adj, f, depth(v), ∞, depth(v))
10:         num_threads ← num_threads + 1
11:     end while
12:     while num_threads > 0 do
13:         wait until a termination signal is received
14:         if L ≠ ∅ then
15:             remove any v ∈ L
16:             RS2(v, Δ, Adj, f, depth(v), ∞, depth(v))
17:         else                                                     ▷ Phase 3
18:             num_threads ← num_threads − 1
19:         end if
20:     end while
21: end procedure
```

After each of the first two threads terminate, in lines 15-16, two further calls are made: $RS2(2143, 3, Adj, f, 2, \infty, 2)$ and $RS2(1423, 3, Adj, f, 2, \infty, 2)$. Then $L = \emptyset$ and each subsequent termination decrements *num_threads* until all threads have completed.

In analyzing Algorithm 3 we observe that in Phase 1 there is no parallelization, in Phase 2 all available cores are used, and in Phase 3 the level of parallelization drops monotonically as threads terminate. Looking at the overhead compared with Algorithm 1 we see that this almostly entirely consists of the ammount of time required to restart the reverse search process. This leads to conflicting issues in setting the critical *init_depth* parameter. A larger value implies that:

- only a single thread is working for a longer time
- the list L will be typically be larger requiring more overhead in restarts, but
- the time spent in Phase 3 will typically be reduced.

The success in parallelization clearly depends on the structure of the tree T. In the worst case it is a path and no parallelization occurs in Phase 2. In the best case the tree is balanced balanced so that the list L can be short reducing overhead and all threads terminate at more or less the same time. Success therefore heavily depends on the structure of the underlying enumeration problem.

For the vertex enumeration problem, discussed in the next section, both of these extremes and everything in between is possible. We will see experimental results to illustrate this in Section 5.

We conclude by comparing the method of Algorithm [**?**] with that used by Weibel [11] for computing Minkowski sums by reverse search. The latter

method uses a more sophisticated approach. Firstly the search is recursive so that all nodes are stored in the backtrack path. As we noted, for vertex enumeration it is not possible in general to keep a full bactrack stack since it may contain all of the LP dictionaries and exhaust memory. An approximation to this included in the original *lrs* code, which employs a user specified parameter k and caches the last k nodes of the backtrack stack. In this way memory is not exhausted and the number of cache misses is usually rather low.

Secondly the rather than executing a distinct Phase 1, in Weibel's method a given process is designated the *boss* and can either execute normally or spin off nodes to other threads to be executed in parallel. When the boss runs out of work another node is designated to be the boss and messages are sent to inform all other nodes.

Computational experience is given for up to 8 parallel processors with reported speedups of 5.5 to 8 times.

3 Vertex enumeration

3.1 Reverse search vertex enumeration method

The initial application of reverse search was to the vertex enumeration problem [3]. From this paper the *lrs* program was derived and a full description of its implementation is given in [1]. We give a simplified description here.

Given an $m \times n$ matrix $A = (a_{ij})$ and an m dimensional vector b, a *convex polyhedron*, or simply *polyhedron*, P is defined as:

$$P = \{x \in R^n : b + Ax \geq 0\}.$$

A *polytope* is a bounded polyhedron. For simplicity in this description we will assume that we are dealing input data A, b that define full dimensional polytopes. A point $x \in P$ is a *vertex* of P iff it is the unique solution to a subset of n inequalities solved as equations. The *vertex enumeration problem* is to output all vertices of a polytope P. Figure 2 shows a typical input which defines the polytope P sketched in Figure 3 with 5 vertices.

$$A = \begin{bmatrix} -1 & 0 & 1 \\ 0 & -1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & -1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \begin{array}{l} 1 - x_1 + x_3 \geq 0 \\ 1 - x_2 + x_3 \geq 0 \\ 1 + x_1 + x_3 \geq 0 \\ 1 + x_2 + x_3 \geq 0 \\ -x_3 \geq 0 \end{array}$$

Fig. 2 A, b and its polyhedron P

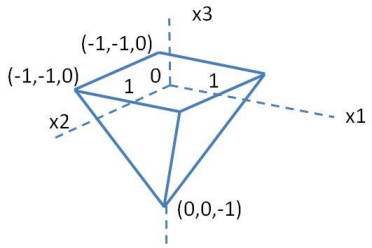

Fig. 3 P has 5 vertices

The computations are based on *dictionaries*, as is done for the simplex method of linear programming. To get a dictionary for $P = \{x \in R^n : b + Ax \geq 0\}$ we add one new nonnegative variable for each inequality:

$$x_{n+i} = b_i + \sum_{j=1}^{n} a_{ij} x_j, \quad x_{n+i} \geq 0 \quad i = 1, 2, ..., m.$$

These new variables are called *slack variables* and the original variables are called *decision variables*.

In order to have any vertex at all we must have $m \geq n$, and normally m is significantly larger than n, allowing us to solve the equations for various sets of variables on the left hand side. The variables on the left hand side of a dictionary are called *basic*, and those on the right hand side are called *non-basic* or, equivalently, *co-basic*. We use the notation $B = \{i : x_i \text{ is basic}\}$ and $N = \{j : x_j \text{ is co-basic}\}$.

A *pivot* interchanges one index from B and N and solves the equations for the new basic variables. A *basic solution* from a dictionary is obtained by setting $x_j = 0$ for all $j \in N$. It is a *basic feasible solution(BFS)* if $x_j \geq 0$ for every slack variable x_j. A dictionary is called *degenerate* if it has a slack basic variable $x_j = 0$. As is well known, each BFS defines a vertex of P and each vertex of P can be represented as one or more (in the case of degeneracy) BFSs. For the example a typical BFS and dictionary are shown in Figure 4.

$$\begin{aligned} \boxed{x_1} &= -1 + x_6 + x_8 \geq 0 \\ \boxed{x_2} &= -1 + x_7 + x_8 \geq 0 \\ \boxed{x_3} &= -x_8 \geq 0 \\ x_4 &= 2 - x_6 + x_8 \geq 0 \\ x_5 &= -2 - x_7 - 2x_8 \geq 0 \end{aligned}$$

Fig. 4 Decision variables are all basic, $N = \{6, 7, 8\}$

To apply reverse search to this problem we first define the relevant graph $G = (V, E)$. Each node in V corresponds to a BFS and is labelled with the cobasic set N. Each edge in E corresponds to a pivot between two BFSs. Formally we may define the adjacency oracle as follows. Let B and N be index sets for the current dictionary. For $i \in B$ and $j \in N$

$$Adj(N, i, j) = \begin{cases} N - j + i \text{ if this gives a feasible dictionary} \\ \emptyset \qquad\qquad\quad \text{otherwise} \end{cases}$$

(The notation $N - j + i$ is used as a convenient shorthand for $N \setminus \{j\} \cup \{i\}$.) For the example the graph G is shown in Figure 5. Observe that the vertex $(0, 0, -1)$ is degenerate and is represented by four cobases. The target v^* for the reverse search is found by solving a linear program over this dictionary with any objective function $z = c^T x$ that defines a unique optimum vertex. We use the objective function z and a non-cycling pivot selection rule to define

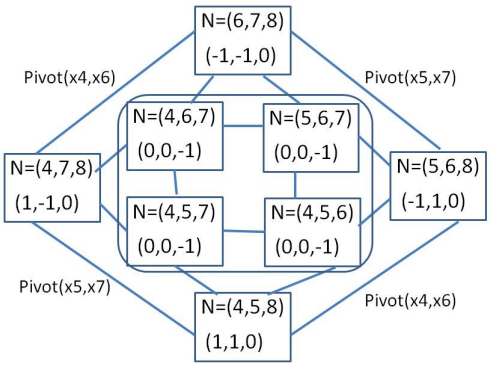

Fig. 5 The graph of feasible dictionaries for Figure 2

the local search function f. In the case of *lrs* we use Bland's least subscript rule for selecting the variable which enters the basis and a lexicographic ratio test to select the leaving variable. This lexicographic rule simulates a simple polytope which greatly reduces degeneracy. In the example only two of the four bases defining vertex $(0, 0, -1)$ would be generated. For details see [1].

3.2 Parallelization issues

In this subsection we discuss issues affecting how successful we can expect Algorithm 3 to be when applied to vertex enumeration. Referring back to the analysis at the end of Section 2.3 we recall the worst case is when the reverse search tree T is a path, as no parallelization is achieved. This in fact can happen!

The Klee-Minty examples [9], and their relatives, are specially constructed polytopes so that the simplex method with a given pivot rule will follow a Hamiltonian path on the polytope's skeleton. This is precisely the case when no parallelization occurs. This creates no problem for single processor codes, as the tree shape is largely irrelevant. Examples of various types of polytopes are given in Section 5.

4 Implementation description

As discussed earlier, the first attempt at parallelization was unfortunately ineffective, unportable and most importantly unmanageable. The approach essentially used POSIX threads to initiate a system call of lrs on subtrees. Concatenating the output to a standard location proved difficult as interprocess communication is not easily achieved. To circumvent this issue, temporary files were used to store output data. This was inefficient with regards to

memory requirements for a given problem. These short comings were carefully reviewed when the parallelization problem was attacked for a second time.

In the second approach, several open source, multi-threading libraries were considered. It was decided that the C++ Boost library offered the greatest performance, adaptability and maintainability. Moreover, Boost works on almost any modern operating system, including UNIX and Windows variants; ensuring the portability of the final solution. Although C++ is not a strict superset of C, the language provides mechanisms for mixing code that is compiled by compatible C and C++ compilers. This allowed us to create a lightweight C++ wrapper around the lrs codebase using the g++ compiler.

On a high level, *plrs* has a multi-producer single consumer architecture. What this equates to is that several producer threads travers subtrees of the vertex enumeration problem, while a single consumer thread concatenates output to a unified location. Note that threads within a process share the same state, and same memory space. This is in contrast to processes which are independent execution units that contain their own state information. This leads to the fact that inter-thread communication is easily achieved.

The Boost.Atomic library is used to coordinate these multiple threads through atomic variables. The implementation makes use of processor-specific instructions where possible and falls back to emulating atomic operations through locking; ensuring the portability of the solution. A lock-free multiple producer single consumer queue is used to maintain output. The specific function compare_exchange_weak is used to post output from the producer threads to the single consumer thread. For more details, please visit the Boost.org web site [5].

The boost::thread class is responsible for launching the consumer thread while the boost::thread_group class is used for launching and managing all producer threads. In order to wait for the execution of all producer threads to finish, the join_all() member function is used. Essentially, this blocks the main process thread from completing until vertex enumeration has exhausted. A similar function, join(), is used on the consumer thread to ensure all output is captured before the completion of the entire process.

5 Numerical experiments

We describe here some experimental results using the plrs code on two computers, *mai12* [1] and *mai64* [2] with respectively 12 and 64 cores and similar processor speeds. We initially benchmarked the two computers by running an *lrs* job (single thread) when the computers were idle and then with increasing load averages. With a load average of 12, *mai12* performed essentially the same as with a load average of one, as one would wish for comparative experiments. In a similar test on *mai64* the performance deteriorated noticeably with high load averages. At load averages of 32, 48, and 64 on *mai64* the processing times

[1] Xeon X5640, 2.66GHz, 12 core, 24GB memory, 60GB hard drive

[2] Opteron 16core 6272 X 4, 2.1GHz, 64 core, 64GB memory, 500GB hard drive

Name	Input			Output		*lrs*			cddr+
	H/V	m	n	V/H	size	bases	depth	secs	secs
mit	H	729	9	4861	196K	1375608	101	809	505
bv7	H	69	57	5040	867K	84707280	17	11851	
perm7	H	127	8	5040	127K	5040	21	0.6	15.0
c30-15	V	30	16	341088	73.8M	319770	14	80	4652
perm10	H	1023	11	3628800	127M	3628800	45	3193	
c40-20	V	40	21	40060020	15.6G	20030010	19	22458	

Table 1 Polyhedra tested: *lrs* ,*cddr+* times on *mai12*

were respectively 1.15, 1.41 and 1.46 times longer then with a load average of one. We therefore restricted tests to a maximum of 32 threads on this machine and, even so, these results probably underestimate the speedup by about 15%. The *mai12* results are more robust.

We chose a few representative polyhedra that are shown in Table 1. For each example we first give the input file name, type (H or V-representation) and input dimensions (m rows and n columns). We then give the output size (number of vertices or facets, respectively) and space, which ranges from 127K to an enormous 15.6G. For *lrs* we give the number of bases generated, the maximum tree depth and running time in seconds. The *cddr+* times were obtained using the default settings and are only given to emphasize the difference between pivoting and double description methods. No attempt was made to optimize the settings. No value implies that the *cddr+* run did not terminate with 48 hours. Input files are available from the web site [2].

The polytope *mit* is a configuration polytope which required about a month of computer time for its vertex enumeration by *cdd* and *lrs* when first run in 1993 [7]. It is a rather degenerate polytope. *c40-20* is a cyclic polytope and is simple, ie, non-degenerate. *perm7* and *perm10* are permutation polytopes written in their standard formulations, and are also simple polytopes. The vertices of *perm_n* are the $n!$ permutations of $1, 2, ..., n$. The standard formulation using n variables has $2^n - 2$ inequalities and one linearity. *bv7* is an alternative formulation that has polynomial size in n as it is based on the Birkhoff-Von Neumann polytope. It has n^2 inequalities and $3n - 1$ linearities in $n^2 + n$ variables. We included *perm7* only for comparison purposes with *bv7* and do not use it in parallelization experiments.

In Tables 2 and 3 we present speedup results for *plrs* runs on *mai12* and *mai64* respectively for the problems presented in Table 1. The initial depth parameter was chosen fairly arbibtrarily to give a reasonable size list L of problems to solve in parallel. On *mai12* we observe that the speedups are roughly comparable except for the last problem, *c40-20*, which are considerably smaller. As remarked, it has huge output size. On *mai64* in addition *c30-15* shows very small speedups as the number of threads increases. Note that it has short running time relative to its output size. On both machines the speedups are largest for the highly degenerate *bv7* which generates very little output. Together this is evidence that the collection process may be the bottleneck in these cases.

Name	*lrs*	mt = 4		mt=8		mt=12	
	secs	secs	su	secs	su	secs	su
		L	id	L	id	L	id
mit	809	232	3.5	142	5.7	104	7.8
		284	4	613	5	1213	6
bv7	11851	3117	3.8	1580	7.5	1104	10.7
		645	2	645	2	7554	3
c30-15	80	27	3.0	15	5.3	12	6.7
		1716	6	1716	6	1716	6
perm10	3193	983	3.2	517	6.2	421	7.6
		4489	7	4489	7	4489	7
c40-20	22458	9633	2.3	5600	4.0	3697	6.1
		220	3	715	4	2002	5

Table 2 Times and speedups (su): no. of threads =mt, initial depth=id (*mai12*)

Name	*lrs*	mt = 4		mt=8		mt=16		mt=32	
	secs	secs	su	secs	su	secs	su	secs	su
		L	id	L	id	L	id	L	id
mit	1125	339	3.3	190	5.9	123	9.1	110	10.2
		284	4	613	5	1213	6	2121	7
bv7	17381	4513	3.85	2345	7.4	1215	14.3	707	24.5
		645	2	645	2	7554	3	7554	3
c30-15	75	34	2.2	22	3.4	20	3.8	21	3.6
		1716	6	1716	6	1716	6	1716	6
perm10	4295	1317	3.3	683	6.3	566	7.6	570	7.6
		4489	7	4489	7	4489	7	4489	7
c40-20	17538	9802	1.8	6707	2.6	4902	3.6	4106	4.3
		220	3	715	4	2002	5	5005	6

Table 3 Times and speedups (su): no. of threads =mt, initial depth=id (*mai64*)

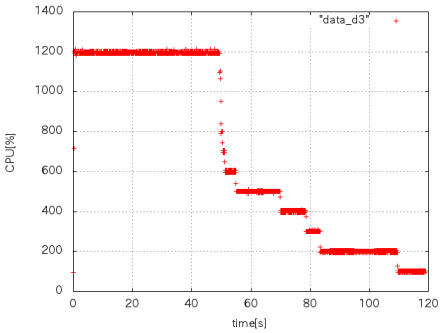

Fig. 6 *mit*: mt=12, id=3, *mai12*

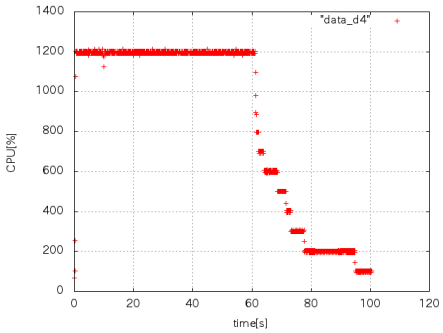

Fig. 7 *mit*: mt=12, id=4, *mai12*

The only user parameter for *plrs* is the initial depth parameter. If this parameter is too low the list of jobs L may be too short to provide adequate parallelism. On the other hand if it is too large a relatively large amount of time will be spent in phase 1 using only one thread, and also in restarting each job in L during phase 2. This is illustrated in Figures 6 to 9. These show the

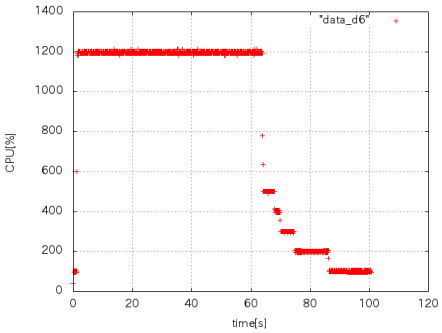

Fig. 8 *mit*: mt=12, id=6, *mai12*

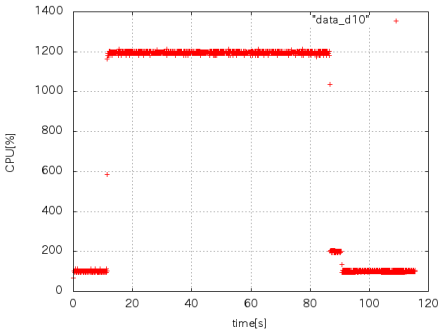

Fig. 9 *mit*: mt=12, id=10, *mai12*

Name	*lrs*	id = 2		id = 3		id=4		id=5		id=6	
	secs	L	secs	L	secs	L	secs	L	secs	L	secs
mit	809	35	183	115	127	284	102	613	104	1213	108
bv7	11851	645	1144	7554	1104	60966	1126	349984	1499	-	-
c30-15	80	36	34	120	23	330	17	792	13	1716	12
perm10	3193	44	405	155	457	440	507	1068	530	2298	551

Table 4 Comparison of various init_depth(id) values with max_threads(mt)=12 (*mai12*)

load average during runs of *plrs* on *mit* using 12 cores and depths 3,4,6, 10 respectively. One can clearly see the three phases of execution: a short startup with one core, a period with all 12 cores active while L is depleted, then the final phase as each process terminates. The area under the graph is the total execution time. Depths of 4 and 6 achieve the fastest elapsed time, but total execution time is higher at depth 6. With depth 10 the first two phases are longer, the third phase shorter, the total execution time longer. However the elapsed time to complete the run is about 20% longer than at depth 4.

In Table 4 we show the dependence on the initial depth of speedup results for *plrs* on four of the polytopes. Although there are differences in performance they are less important than we expected. Given the previous discussion, one would expect increasing speedups as the depth increases from a small value to a minimum then decreasing speedups as the depth increases. Although this is somewhat observed in the data there are obviously other competing factors at play and the situation is more complicated.

6 Conclusions and future work

We have demonstrated that very useful speedups can be obtained by a portable parallelization of *lrs* that does not disturb the underlying code. The method allows for independent development of the *lrs* code and the parallelization process itself. We expect that similar results can be obtained by a wide range of applications using the reverse search approach. The installation is straight

forward and no special purpose hardware is required. Very noticeable improvements are found using just quad-core personal computers.

Figure 7 shows the limitations of our approach. For the polytope *mit* an initial depth of 4 achieves the shortest elapsed time. However for only 60% of the time are all 12 cores busy. In fact for a quarter of the time only two cores are active.

To rememedy this one can imagine interupting long running tasks and then using plrs recursively to split them into subproblems, repopulating L. We performed some preliminary experiments along these lines, but the results were mixed. As this increases overhead, the final result may sometimes be worse, depending on the search tree shape. It is a fruitful area for future research. Another possibility is to use the built in estimator function of *lrs* . For each leaf obtained in phase 1 it is possible to get an unbiased estimate of the size of the subtree that it roots by using a random probe. One could then schedule jobs from L using a list decreasing heuristic, so that longer runs are done first. The tradeoff is again overhead: the random probes may require a lot of processing time if the tree is unbalanced.

7 Acknowledgments

The authors would like to thank Kenji Okuda for preparing Figures 6 to 9.

References

1. Avis, D.: lrs: A Revised Implementation of the Reverse Search Vertex Enumeration Algorithm. In: G. Kalai, G. Ziegler (eds.) Polytopes - Combinatorics and Computation, pp. 177–198. Springer (2000)
2. Avis, D.: (2013). http://cgm.cs.mcgill.ca/~avis/C/lrs.html
3. Avis, D., Fukuda, K.: A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. Discrete & Computational Geometry **8**, 295–313 (1992)
4. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Applied Mathematics **65**, 21–46 (1993)
5. Boost.org: (2013). http://www.boost.org/doc/libs/1_53_0/doc/html/lockfree.html
6. Brungger, A., Marzetta, A., Fukuda, K., Nievergelt, J.: The parallel search bench ZRAM and its applications. Ann. Oper. Res. **90**, 45–63 (1999)
7. Ceder, G., Garbulsky, G., Avis, D., Fukuda, K.: Ground states of a ternary fcc lattice model with nearest- and next-nearest-neighbor interactions. Phys Rev B Condens Matter **49**(1), 1–7 (1994)
8. Fukuda, K.: (2012). http://www.inf.ethz.ch/personal/fukudak/cdd_home
9. Klee, V., Minty, G.J.: How Good is the Simplex Algorithm? In: O. Shisha (ed.) Inequalities III, pp. 159–175. Academic Press Inc., New York (1972)
10. Marzetta, A.: (2008). Maintained by D. Bremner: http://www.cs.unb.ca/~bremner/software/zram/
11. Weibel, C.: Implementation and parallelization of a reverse-search algorithm for minkowski sums. In: ALENEX, pp. 34–42 (2010)