

Implementation and Parallelization of a Reverse-Search Algorithm for Minkowski Sums

Christophe Weibel*

Abstract

We present an implementation of a reverse-search algorithm of Fukuda for computing Minkowski sums of polytopes efficiently. The algorithm allows summing any number of polytopes in any dimension, and is complete in the sense that it does not assume general position. Its running time depends linearly on the size of the output. To the best of our knowledge, this is the only existing implementation that can efficiently compute Minkowski sums in higher dimensions. The implementation uses the exact arithmetic GMP, which ensures robustness of the program and exactness of the results. We furthermore present a parallel version of our implementation to demonstrate the simplicity and efficiency of performing the reverse search in parallel. The results of the performance tests show a near-linear acceleration of our parallel implementation.

1 Introduction

Given two polytopes P_1 and P_2 , their *Minkowski sum* is defined as $P = P_1 + P_2 = \{x = x_1 + x_2 : x_1 \in P_1, x_2 \in P_2\}$. This definition naturally extends to sums of multiple polytopes.

Minkowski sums of 3-dimensional polytopes have long been known to have applications in motion planning [11]. More recently, applications have been found for sums of many polytopes in higher dimensions in fields as diverse as algebraic statistics [10] and decision processes [16].

In this article, we present an implementation that computes Minkowski sums of polytopes in higher dimensions efficiently. To the best of our knowledge, our implementation is the only one in that respect, and has already been used in various domains [3] [13]. Note that in dimension 3, there are other efficient algorithms for computing Minkowski sums (e.g. [6] [9]). Some of the implementations based on those algorithms have been compared to ours in [5] and [6]. Among those implementations, the fastest is that of Fogel and Halperin [6], based on overlays of Gaussian maps. However, the algorithms used by those implementations cannot easily

be extended to dimensions higher than 3.

Our implementation is based on an algorithm of Fukuda [7], which uses the reverse search method presented by Avis and Fukuda in [1]. The algorithm is different from others in that it only computes vertices of the sum, whereas other implementations compute facets of the Minkowski sum. In general dimension, the time necessary to compute the facets of a Minkowski sum from the facets of the summands is exponential in terms of the input and the output, unless P is NP [15]. In contrast, the algorithm we use is polynomial in the size of the input and *linear* in the size of the output.

However, the time necessary can still be very large, as the size of the output can be exponential in the size of the input. For instance, the sum of d linearly independent segments in \mathbb{R}^d ($2d$ vertices) is equivalent to a d -cube (2^d vertices). This is why we have developed a simple parallel implementation of the algorithm, taking advantage of the fact that the reverse search method can be parallelized very efficiently. Indeed, implementing a reverse search algorithm in parallel is relatively easy, and provides significant benefits. Our performance studies show that the parallel implementation provides an almost ideal speedup when computing simple cases, and a more moderate but still impressive one in more complex instances.

The rest of the paper is organized as follows. We start with an introduction to relevant properties of Minkowski sums in Section 2. We present in Section 3 the reverse search method and its advantages. In Section 4, we provide some details about our implementations, and in particular the parallel version. We present our performance study in Section 5 and finally conclude in Section 6.

2 Minkowski sums of polytopes

Let P_1, \dots, P_r be polytopes. Their Minkowski sum is the polytope defined as $P_1 + \dots + P_r = \{x_1 + \dots + x_r : x_i \in P_i \forall i\}$.

A *face* of a polytope P is the subset of the polytope on which some linear function is maximized. Vertices are faces of dimension 0, edges are faces of dimension 1, and facets are faces of dimension one less than P .

*Department of Mathematics and Statistics, McGill University, 805 Sherbrooke West, Montreal, QC, Canada. Email: christophe.weibel@gmail.com

Each face F of a polytope P is associated with its *normal cone*, which is the set of linear functions on the polytope which are maximised on this face: $\mathcal{N}(F; P) = \{l : l^t x \geq l^t y, \forall x \in F, y \in P\}$. The normal cone is an open polyhedral cone in the adjoint space of linear functions defined over \mathbb{R}^d , pointed at the origin. The normal cones of all faces of a polytope determine a subdivision of the adjoint space into polyhedral cones, called the *normal fan*: $\mathcal{N}(P) = \{\mathcal{N}(F; P) : F \text{ face of } P\}$.

In the Minkowski sum $P = P_1 + \dots + P_r$, let F be a face and l a linear function in the corresponding normal cone $\mathcal{N}(F; P)$. Any point x in F is a sum of points $x_1 + \dots + x_r$ in the summands. Since x maximizes $l^t x$ over P , each x_i must also be a maximizer of the function l in the polytope P_i ; if there was a $y_i \in P_i$ with $l^t y_i > l^t x_i$, then x would not be a maximum for l , because $x - x_i + y_i$ would have a higher value. Therefore, a face of P maximizing a linear function l can be decomposed into a sum of faces of the summands, each maximizing l in their respective polytopes.

As a consequence, the normal cone of a face of the Minkowski sum is a set of linear functions which are maximized on the same faces for each summand, i.e. a nonempty intersection of normal cones of the summands. The normal fan of the Minkowski sum thus defined is said to be the *common refinement* of that of its summands:

$$\mathcal{N}(P_1 + \dots + P_r) = \{\mathcal{N}(F_1; P_1) \cap \dots \cap \mathcal{N}(F_r; P_r) : F_i \text{ face of } P_i\}.$$

Vertices of a Minkowski sum decompose into vertices of the summands. Edges decompose into vertices and parallel edges of the summands. Most importantly, if two vertices x and x' are adjacent in P , and are decomposed into $x_1 + \dots + x_r$ and $x'_1 + \dots + x'_r$ respectively, then for all i in $1, \dots, r$, either $x_i = x'_i$, or x_i and x'_i are adjacent to each other, and the edge of P_i incident to x_i and x'_i is parallel to the edge of P incident to x and x' .

Thus, we can find vertices adjacent to a vertex of the sum by examining all edges incident to the vertices in their decomposition, and finding out which correspond to an incident edge of the sum.

3 The reverse search algorithm

In this section, we give a succinct description of the reverse search algorithm that we implemented. This algorithm, introduced by Fukuda in [7], computes the vertices of a Minkowski sum of polytopes. It can be viewed as a direct application of the reverse search method first introduced by Avis and Fukuda in [1].

3.1 The reverse search method The reverse search method is used to enumerate a large set of objects that are arranged in a graph $G(V, E)$, where the set of nodes V is the set of objects, and the edges E can be deduced using an *adjacency oracle*, which allows us to enumerate the neighbours of a node in the graph. If properly implemented, the reverse search takes a time linear in the size of the graph.

We define an arborescence on this graph, that is, a tree of oriented edges with a single sink. As a consequence, any node except the sink has a unique outgoing incident arc, which can be found using a *local search function*. Thus, from any vertex, we can find the sink by using the local search function repeatedly. Examples of local searches include the simplex algorithm looking for an optimal basis, or the flip algorithm looking for a Delaunay triangulation. For any node, the adjacent node indicated by the local search function is its *parent*. Conversely, a *child* of a node is any adjacent node of which it is the parent.

As the name implies, reverse search is a reversal of this local search, starting from the sink and finding all vertices. The arborescence is explored depth-first. For any new node N encountered, the algorithm searches through neighbours of N for its children, using the adjacency oracle. Children of N can be identified by the fact that their parent, found by the local search function, is N . In the simplest of implementations, the branching rooted at the child is enumerated recursively:

```

procedure explore( $N$ )
  for each neighbour  $C$  of  $N$ 
    if parent of  $C$  is  $N$ 
      explore( $C$ )
  print  $N$ 

```

This differs from the classical reverse search method defined in [1] as follows. By using a recursive function, the program keeps in the memory stack the variables for each call to the recursive function; there are as many recursive calls as the depth of the arborescence, which could technically lead to a memory usage linear in the size of the output. In a classical reverse search method, this is implemented without recursion so as to ensure that the memory usage is independent from the output size. However, this results in more calls to the local search function to find the way back (See [1], [7] for details). The results in Section 5.2 show that in the case of Minkowski sums, the recursive version is faster without requiring significantly more memory.

The strength of the reverse search method is that nodes that have been enumerated and printed can be forgotten, and therefore need not be kept in memory for further comparisons, as a standard enumeration would.

This is especially important in the case of Minkowski sums, as the output size can be very large even for small inputs. Another advantage of this is that, should a computation be interrupted for some reason, it is easy to restart at the last node that was printed to file. Finally, the exploration of different branches of the arborescence can be easily done in parallel, since each exploration is independent. The ZRAM parallel search library offers an interface for executing a reverse search in parallel [2]. However, it does not allow the faster recursive version of reverse search, we therefore did not use it.

3.2 Minkowski sums As stated, our implementation is based on an algorithm published, with details of the adjacency oracle and of the local search function, by Fukuda in [7].

The graph enumerated by the reverse search is that of vertices and edges of the Minkowski sum. A first vertex of the sum can easily be found by choosing a linear function which has a unique maximum vertex on all summands, and taking the sum of these vertices.

In order to find edges of the graph, the algorithm uses the fact that an edge incident to a vertex of the sum is parallel to some edge incident to some vertex in the decomposition of the vertex. Edges of the sum can thus be found by examining edges from the summands. To speed up the algorithm, the edges of the summands are precomputed before the beginning of the reverse search. We defer the details of the adjacency oracle and of the local search function to Appendix A.1.

The complexity of the algorithm is $O(\delta LP(d, \delta)|V|)$, where δ is the sum over summand polytopes of the maximum degree of their vertices, d is the dimension of the Minkowski sum, $LP(m, n)$ is the time necessary to solve a linear program of m inequalities in dimension n , and $|V|$ is the size of the output.

More precisely, for each vertex v of the Minkowski sum, let $\delta(v)$ be the number of edges incident to vertices in the decomposition of v . In order to find the edges incident to v , the adjacency oracle solves $\delta(v)$ linear programs of $\delta(v)$ inequalities in dimension d .

Also, for any adjacent vertex x found by the adjacency oracle, the local search function solves a linear program of $\delta(x)$ inequalities in dimension d . This corresponds to twice as many linear programs as there are edges in the sum. However, arcs of the arborescence are oriented consistently with a known linear function. By testing the direction of the edge, it is possible to use the local search function in only half of the cases, which results in one linear program per edge of the sum.

The resolution of linear programs for the adjacency oracle and the local search function dominate the complexity of the algorithm.

4 Implementation

The algorithm was implemented in C++, using the exact arithmetic library GMP [4]. We also make use of various functions in Komei Fukuda's CDDLIB library [8], in particular for the resolution of linear programs. The general organization of the implementation, as well as many of the C++ data structures used for manipulating geometrical objects that are defined on GMP rationals, were adapted from the triangulation package TOPCOM [14], written by Jörg Rambau, to whom we are very thankful.

The Gnu Multiple Precision Arithmetic Library (GMP) offers the rational number type `Gmpq`, over which the four basic operations can be done with unlimited precision. This ensures the robustness of our implementation and correctness of the results.

We implemented both a recursive version of the reverse search, and a classical reverse search, whose memory usage is independent of the output size. As these two implementations are fairly straightforward, we will not go into further details. We focus instead on the parallel implementation, so as to give a clear picture of how simply and efficiently the reverse search can be parallelized.

4.1 Parallel implementation The parallel implementation is based on a recursive implementation of the reverse search, as described in Section 3. Recall that in the sequential implementation, whenever the process, while enumerating the neighbours of a vertex, finds a child of that vertex, it immediately explores the branching rooted at that child before looking for other children of the vertex, as in a classical depth-first search. But in the parallel implementation, a process discovering a child may instead assign to another process the task of exploring the branching rooted at the child, and continue looking for other children of the same vertex.

The advantage of separating the workload this way is that the exploration of a branching of the arborescence needs no information from any other part of the enumeration, except for the emplacement of the root. Branchings offer therefore a very natural division of work between processes. The transmission of data between processes is minimal, and the parallelization requires little modification to the sequential implementation. As a matter of fact, the parallel implementation added less than 200 lines of code to the sequential one.

In the following sections, we explain how branchings to be explored are allocated to different processes. We first introduce in Section 4.1.1 and 4.1.2 the boss process and the messages sent between processes, before explaining the flow of the execution in Section 4.1.3.

4.1.1 Boss process One of the hardest problems of parallelization is to properly balance the load between different processes. Rather than trying to divide the workload as equally as possible, we focus our efforts on allowing any process that runs out of work to find a new task as fast as possible.

The task of distributing work is given to a particular process, called the *boss process*. Though the boss process may change, there is only one boss process at any given time. The first boss process starts the reverse search at the sink, and receives requests for work from other processes.

Whenever the boss process, searching through the neighbours of a vertex, finds a child vertex, it first checks whether it received requests for work from other processes. If so, it sends back the child vertex to the processor that sent the request, and continues searching for more child vertices. Otherwise, it explores itself the child vertex recursively.

When the boss process runs out of vertices to enumerate, it checks which other processes requested for work. If they all did, then the enumeration is over. The boss process sends a termination message to all other processes, and terminates itself. If however some process did not send a request for work, then that process is still enumerating some branching, and so it can become the next boss. The current boss sends a message to all other processes to inform them of the change.

Because they might be busy, some processes learn later than others the change of boss. However, at most one process considers itself to be the boss at any time, as no process except the boss can decide the next one, and the boss stops seeing itself as so at the instant it decides the next one.

Since many boss changes can happen before a process learns of any of them, each process keeps a *change number*. Requests for work and boss change announcements are marked with the change number of the sender, and are ignored if it is lower than that of the receiver.

4.1.2 Messages sent between processes There are four types of messages sent between processes.

1. The *request for work*. It is sent from an idle process to the boss process, asking for a branching to explore.
2. The *task notice*. It is sent from the boss process to another process that sent a request for work. This message indicates the vertex to explore.
3. The *boss change announcement*. It is sent to all processes by the boss process, when it has run out

of work and has found another processor that is still working. This message indicates the new boss.

4. The *termination signal*. It is sent to all processes by the boss process, when it has run out of work and all other processes have requested for work. This message ends all processes.

We explain in Appendix A.3 the communications between processes and the reaction of processes to messages in more detail.

4.1.3 Flow of the execution As shown in the following pseudocode, the general flow of the execution is based on the recursive implementation of the reverse search, except that the exploration of the branching rooted at a child node could be assigned to a different process.

```

procedure explore( $N$ )
  for each neighbour  $C$  of  $N$ 
    if parent of  $C$  is  $N$ 
      assign_or_explore( $C$ )
  print  $N$ 
  
```

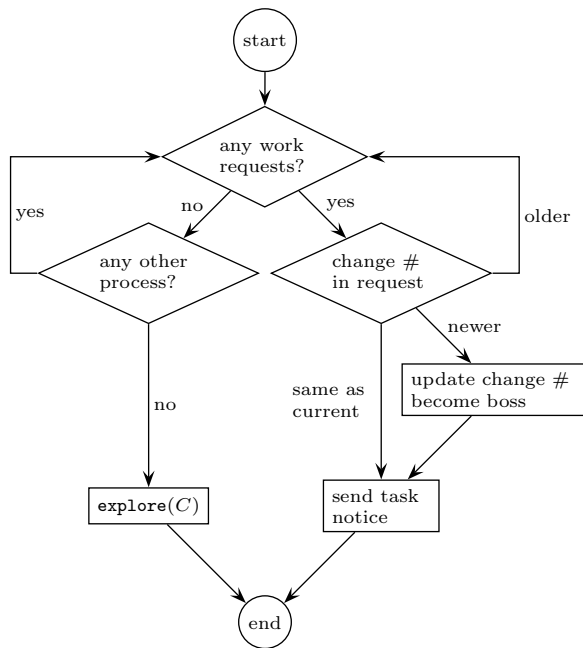


Figure 1: Flowchart of the `assign_or_explore` procedure

The `assign_or_explore` procedure consists of dealing with the task of exploring the branching rooted at the child C , by either performing it, or assigning the task to another process that sent a work request. We present the flowchart of this procedure in Figure 1.

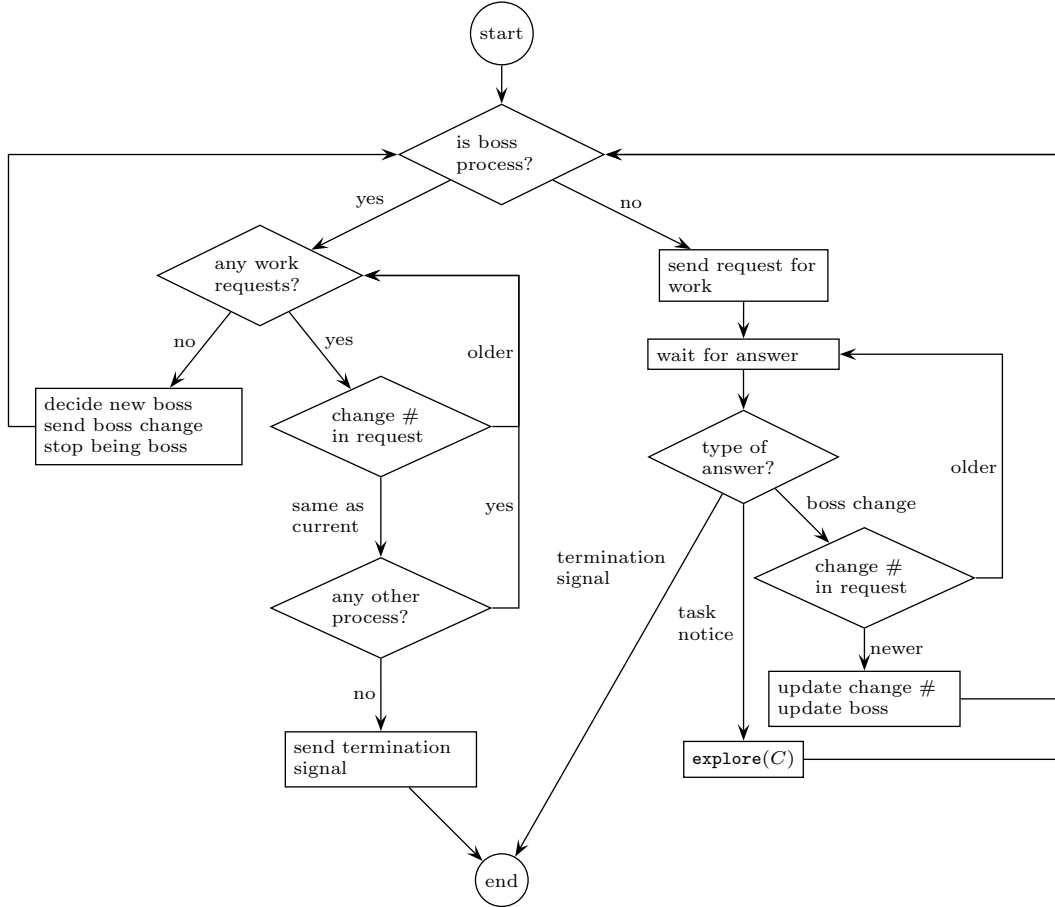


Figure 2: Flowchart of the `idle` procedure

After the initialization, the main procedure can be resumed like this:

```

procedure main()
  if boss
    explore(sink)
  idle()

```

That is, the boss process starts the exploration of the tree at the sink, and all other processes call the procedure `idle`, which lets processes ask for work to the boss process. Once the boss has distributed all branchings, it then also calls the procedure `idle`. Usually, other processes are still computing the last branching they were given. The boss decides its successor, which will distribute work to other processes. When all processes have finished their branchings, the current boss sends to all the message to quit the procedure. We present the flowchart of the procedure `idle` in Figure 2.

5 Performance study

We present in this section the time performance of the implementations for various instances. We first describe the settings of the experiments, then discuss the results. Recall that in dimension 3, there are other implementations for computing Minkowski sums that are faster than ours. We emphasize therefore the performance of our implementation in higher dimensions.

5.1 Experiments

We design two suites of experiments for the performance study.

The first suite of experiments is designed to compare the running time of the algorithm with the output size. Given that the running time is also dependent on the size of linear programs that are solved, we design the instances so that linear programs solved have the same size across instances. We achieve this by comparing sums of n polytopes, k -regular, in dimension d ; so each of the linear programs described in Section 3.2 that is involved in the computation has nk inequalities in di-

cpu	20-cube init. time: 0 2 ²⁰ vertices		10-cube init. time: 0 1024 vert.		HMM6 init. time: 38 17354 vert.		HMM5 init. time: 9 5266 vert.		R(5, 200, 4) init. time: 17 9255 vert.		R(2, 40, 10) init. time: 13 1463 vert.	
	t	s	t	s	t	s	t	s	t	s	t	s
	1	156562	1	14.1	1	21897	1	1643	1	698	1	978
2	78323	2	7.1	1.99	11184	1.96	826	1.99	351	1.99	541	1.81
3	52240	3	4.8	2.93	7383	2.97	555	2.96	238	2.93	367	2.66
4	39283	3.99	3.6	3.88	5556	3.94	424	3.88	180	3.88	274	3.57
5	31414	4.98	3	4.67	4644	4.72	350	4.69	143	4.88	225	4.35
6	26276	5.96	2.5	5.54	3979	5.5	285	5.76	121	5.77	184	5.31
7	22503	6.96	2.1	6.57	3504	6.25	256	6.42	103	6.78	157	6.23
8	19711	7.94	1.9	7.27	3208	6.83	236	6.96	94	7.43	141	6.94

Table 1: Running times in seconds (t) and speedup (s) of the parallel implementation, for different instances, using from one to eight processors (CPU). The running times do not include the initialization time (indicated in seconds), which is not parallelized.

mension d . We study three series of instances composed of sums of segments, known as *zonotopes*, for three different values of n and d . That is, each instance is a sum of n segments with 0, 1 coordinates in dimension d . To ensure full-dimensionality of the sum, the first d segments form a standard basis. The other segments have random coordinates. As the input segments have random linear dependencies, this creates instances whose output sizes vary, but each linear program solved has the same size.

The second suite of experiments is designed to study the efficiency of the parallel implementation, by computing each instance with different numbers of processors. In order to study the speedup in different situations, we used different types of instances. We chose the sum of 20 orthogonal segments, computing the 20-cube, and the sum of 10 orthogonal segments, computing the 10-cube. We then chose two sums arising in the subject of Hidden Markov Models, HMM5 and HMM6; these are sums of 32, respectively 64, polytopes in dimension 8 (see Section 6.2 of [12] on this subject). Finally, we created two sums $R(m, n, d)$ of m random polytopes, each generated as the convex hull of n random points uniformly distributed in a d -cube. The first is $R(5, 200, 4)$, which is a sum of 5 polytopes in dimension 4 with about 80 vertices each. The second is $R(2, 40, 10)$, which is a sum of 2 polytopes in dimension 10 with 40 vertices each.

5.2 Results The first experiment was conducted on a computer with four CPU clocked at 2.3 Ghz, using the sequential classical reverse search implementation and the sequential recursive reverse search implementation. The results are shown in Figure 3(a). As we can observe, the running times are approximately linear in the size

of the output, as predicted by the complexity of the algorithm. The classical reverse search can be seen to be approximately 10% slower than the recursive version. Note that the number of linear programs solved is also dependent on the number of edges in the Minkowski sum. Though the number of edges has an upper bound that is linear in the number of vertices ($O(\delta|V|)$), it shows superlinear growth in the range of our inputs.

The second experiment was conducted on a computer with eight CPU clocked at 2.66 Ghz, using the parallel implementation and compared to the sequential recursive implementation. The results of the second experiment are shown in Table 1, and the speedup is represented in Figure 3(b). The speedup indicates how much faster the parallel version is than the sequential one, depending on the number of CPU used. Note that the initialization of the algorithm, which is essentially spent computing the edges of input polytopes, is not parallelized. The initialization time is therefore omitted from the running times, in order to properly compute the speedup. We can observe that the largest instances show the best speedup, as is generally the case. In particular, as shown in Figure 3(b), the computation of the 20-cube shows nearly perfect speedup. It should also be noted that the speedup with one processor is 1; that is, the parallel version is as fast as the sequential one, even with one processor, which implies there is no additional running time cost on the parallelization.

We can see from Figure 3(b) that the speedup is not always perfectly linear. This can be attributed to the time spent on waiting for an answer from the boss process. For instance, the instance HMM6 shows a speedup comparatively low for its size; but it is the only instance which spends more than one second per vertex. In consequence, the boss process takes a longer

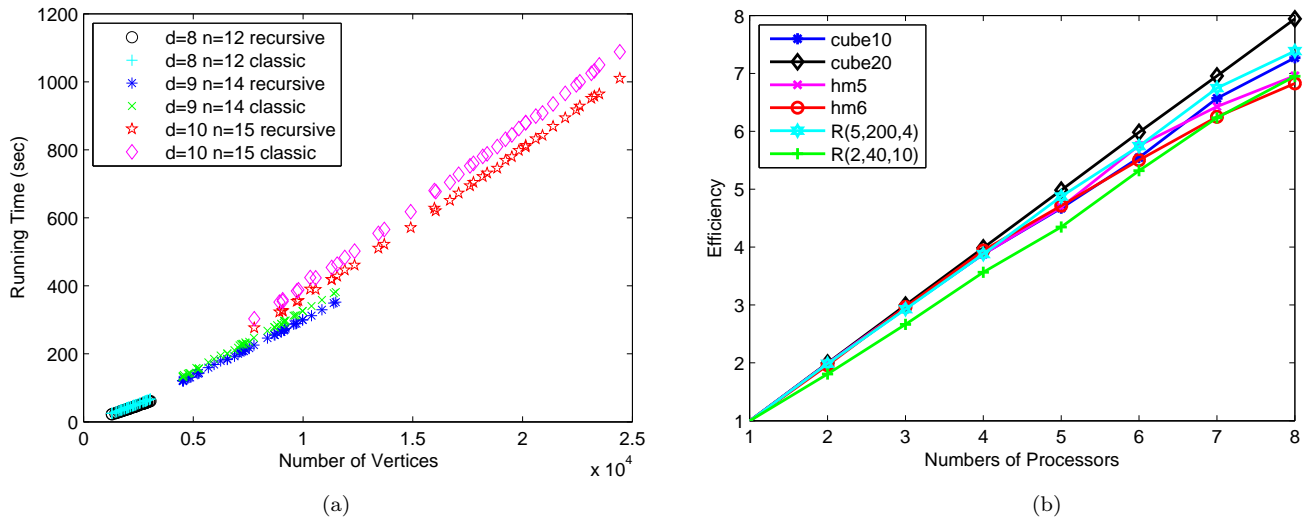


Figure 3: (a) Running time of the classical and recursive reverse search implementations, with relation to the number of vertices. In each series, the instances have different output size, but the linear programs solved have the same dimensions. (b) Speedup of the parallel implementation for different instances.

time to answer to requests for work. This is caused by the roughness of the workload distribution scheme in the current implementation. There are different methods that could alleviate this, but which we did not have time to study. We could for instance have the boss process build up a pile of tasks while other processes are busy, then check more often for task requests and allocate tasks from the pile. We could also have each process manage a list of tasks.

6 Conclusion

Our results show that reverse search offers an efficient way to compute Minkowski sums of polytopes in higher dimensions. Furthermore, they demonstrate that reverse search algorithms can be parallelized easily, and achieve a very good speedup. The performance results of our implementations can also serve as reference to those who are interested in computing Minkowski sums, or using our implementations. The source code for the sequential and parallel implementations can be downloaded on the web from the following address: <http://www.math.mcgill.ca/~weibel/minksum.php>.

References

[1] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics. The Journal of Combinatorial Algorithms, Informatics and Computa-*

tional Sciences, 65(1-3):21–46, 1996. First International Colloquium on Graphs and Optimization (GOI), 1992 (Grimentz).

- [2] A. Brügger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench ZRAM and its applications. *Annals of Operations Research*, 90:45–63, 1999.
- [3] A. Deza and F. Xie. Hyperplane arrangements with large average diameter. In *Polyhedral Computation*, volume 48 of *CRM Proceedings and Lecture Notes*, 2009.
- [4] T. G. et al. GNU multiple precision arithmetic library. <http://gmp.org/>.
- [5] E. Fogel. *Minkowski Sum Construction and other Applications of Arrangements of Geodesic Arcs on the Sphere*. PhD thesis, Tel-Aviv University, Tel-Aviv, 2009.
- [6] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums of convex polyhedra with applications. In *Proceedings of the eighth Workshop Algorithm Engineering and Experiments (ALENEX'06)*, pages 3–15, 2006.
- [7] K. Fukuda. From the zonotope construction to the Minkowski addition of convex polytopes. *Journal of Symbolic Computation*, 38(4):1261–1272, 2004.
- [8] K. Fukuda. *Cdd and cddplus home page*. Mathematics institute, ETH Zurich, 2007. http://www.ifor.math.ethz.ch/~fukuda/cdd_home.
- [9] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel. Boolean operations on 3d selective nef complexes – data structure, algorithms, and implementation. In *Proceedings of the eleventh*

Annual European Symposium on Algorithms, Vol 2832 of LNCS, pages 654–666. Springer, 2003.

- [10] P. Gritzmann and B. Sturmfels. Minkowski addition of polytopes: computational complexity and applications to Gröbner bases. *SIAM Journal on Discrete Mathematics*, 6(2):246–269, 1993.
- [11] T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.
- [12] L. Pachter and B. Sturmfels, editors. *Algebraic statistics for computational biology*. Cambridge University Press, New York, 2005.
- [13] J.-P. Petit. *Spécification géométrique des produits : Méthode de détermination des tolérances. Application en conception assistée par ordinateur*. PhD thesis, Université de Savoie, 2004.
- [14] J. Rambau. TOPCOM: Triangulations of point configurations and oriented matroids. In A. M. Cohen, X.-S. Gao, and N. Takayama, editors, *Mathematical Software—ICMS 2002*, pages 330–340. World Scientific, 2002.
- [15] H. R. Tiwary. On the hardness of minkowski addition and related operations. In *SCG '07: Proceedings of the twenty-third annual symposium on Computational geometry*, pages 306–309, New York, NY, USA, 2007. ACM.
- [16] H. Zhang. Partially Observable Markov Decision Processes: A Geometric Technique and Analysis. *Operations Research*, doi:10.1287/opre.1090.0697, 2009.

A Appendix

A.1 Details of the algorithm We present here for completeness the adjacency oracle and the local search function used in the algorithm presented by Fukuda in [7].

The goal of the adjacency oracle is to enumerate, for any vertex v of the Minkowski sum, all its adjacent vertices. For that, we examine edges incident to vertices in the decomposition of v . The vectors parallel to these edges generate a polyhedral cone. Vectors which are extremal in the cone correspond to edges incident to v in the sum. The extremality can be tested by a linear program. Once an edge e incident to v is found, the vertex at the other end of e can be found by replacing, in the decomposition of v , any vertex which has an incident edge parallel to e by the vertex on the other end.

Let a certain vertex of the sum be called the sink. The goal of the local search function is to find, for any vertex of the sum except the sink, an adjacent vertex called parent, such that a repeated application of the local search finds the sink. If we orient any edge linking a vertex and its parent towards the parent, the edges thus oriented form an arborescence rooted at the sink. To find the parent of a vertex v , we compute a linear

function l_v in the normal cone of v , and a linear function l_s in the normal cone of the sink. Each can be done by solving a linear program. We consider l_v and l_s as vectors in the adjoint space. The adjoint space is subdivided into the normal fan of the Minkowski sum, with l_v and l_s in the normal cones of v and of the sink respectively. If we shoot a ray from l_v to l_s , the ray will hit a facet of the normal cone of v . This facet is the normal cone of an edge incident to v . We choose this edge to be the one linking v to its parent. The local search function thus defined creates no cycles, because the value of the function l_s is always lower at a vertex than at its parent.

In order to execute the adjacency oracle and the local search function efficiently, we start by computing the adjacency matrix of the summand polytopes, which is done by using the `dd_Matrix2Adjacency` function of CDDLIB. Linear programs are solved with CDDLIB's `dd_LPSolve` function. The local search function also uses CDDLIB's `dd_rayShooting` function for shooting a ray from l_v to l_s .

A.2 Technical details of parallelization Due to the simplicity of the parallelization process, we did not need to use any parallel API such as MPI. After initialization is done, which mainly consists in computing the adjacency matrix for each input polytope, additional processes are created using the `fork` system call. This command creates exact copies of the main process. The child processes do not share any memory space with the parent process, but as the common memory only consists of the description and adjacency matrix of each input polytope, the gain of a shared memory system would be negligible.

The processes communicate through pipelines, created with the `pipe` system call before the `fork`. Pipelines can be used to send information from one process to another. In our case, two pipes are created from each process to each other process. One of these two pipes is set to be *blocking*, which means that a process that attempts to read a message from the pipe will pause all activities until it receives one. This is the pipe that will be used to send work, or information on where to find work, to an idle process. Only idle processes will attempt to read from it. The other pipe is set to be *non-blocking*, which means that the receiving process will keep working after attempting to read from the pipe even if it did not receive a message. This is the pipe that will be used by an idle process to ask for work to an active process. Only active processes will attempt to read from it. Pipelines are a standard Unix method for interprocess communication. In consequence, they are normally very fast and efficient.

A.3 Communication between processes We present here in more detail the messages exchanged between processes, and the way they are handled by the processes.

1. The *request for work*. This is sent on the nonblocking pipe from a process that is not the boss to the process it believes to be the boss. Note that the actual boss may have changed. The message contains the change number according to the sender. The sender waits for the answer from the receiver on the blocking pipe.

- If the boss has already changed, the change number in the message is lower than that of the receiver. In that case, the receiver has already sent an announcement informing the sender of the next boss, and the sender reads the announcement immediately after sending the request. The request is therefore ignored.
- If the boss has not changed, then the receiver is the boss. It will send some work when it finds some; if it finds no work, it will send either the boss change announcement if it decides a new boss, or a termination message.
 - Also, if the receiver has a change number lower than that in the message, the receiver learns at that moment that it is the new boss and updates its own change number.

2. The *task notice*. This is sent on the blocking pipe from the boss process to another process that sent a request for work with a change number not lower than that of the boss process. The message indicates the vertex to explore. The process who sent the request is waiting for the answer on the blocking pipe. Requests with a lower change number are ignored, because the boss has changed in the meantime, and a boss change announcement has already been sent to the process which requested for work.

3. The *boss change announcement*. This is sent on the blocking pipe from the boss process to all other processes when it has run out of work and has found another processor that is still working. The message indicates the new boss, and contains the boss change number. If the receiver is idle, it is waiting for this message on the blocking pipe. If it is active, it might learn from a request for work that it has become boss before reading this message. If this happens, the change number of the receiver will be higher than the one in the message,

and the announcement will be ignored when it is read. In this latter case, it is not necessary for the receiver to send another request for work, because it has already sent a request with the higher change number.

4. The *termination signal*. This is sent on the blocking pipe from the boss process to all other processes, when it has run out of work, and all other processes have requested for work with the appropriate boss number. All processes are therefore idle, and waiting for this message on the blocking pipe.