Parametric Search Made Practical

René van Oostrum

Remco C. Veltkamp

Parametric Search Made Practical[*]

René van Oostrum and Remco C. Veltkamp

December 2002

Abstract

In this paper we show that in sorting-based applications of parametric search, Quicksort can replace the parallel sorting algorithms that are usually advocated. Because of the simplicity of Quicksort, this may lead to applications of parametric search that are not only efficient in theory, but in practice as well. Also, we argue that Cole's optimization of certain parametric-search algorithms may be unnecessary under realistic assumptions about the input. Furthermore, we present a generic, flexible, and easy-to-use framework that greatly simplifies the implementation of algorithms based on parametric search. We use our framework to implement an algorithm that solves the Fréchet-distance problem. The implementation based on parametric search is faster than the binary-search approach that is often suggested as a practical replacement for the parametric-search technique.

1 Introduction

Since the late 1980s, *parametric search*, the optimization technique developed by Megiddo in the late 1970s and early 1980s [15, 16], has become an important tool for solving many geometric optimization queries efficiently. The main principle of parametric search is to compute a value λ^* that *optimizes* an objective function f with the use of an algorithm A_s that solves the corresponding *decision problem*. The decision problem can be stated as follows: given a value λ, decide whether $\lambda < \lambda^*$, $\lambda = \lambda^*$, or $\lambda > \lambda^*$. The idea is to run a "generic" version of A_s on the unknown value λ^*. This generic algorithm uses the concrete version of A_s to determine the outcome of the decision problem for a set of concrete values; for one of these values λ, A_s will report that $\lambda = \lambda^*$. We will explain the technique in more detail in Section 2. Usually, applying the concrete version A_s is expensive in terms of running time. Megiddo shows how using a parallel version A_p of A_s as the generic algorithm may reduce the number of times A_s is called considerably.

The technique is rather complicated, as it requires the design of an efficient parallel algorithm. Fortunately, the generic algorithm does not necessarily have to solve the same problem as the concrete decision; in several cases, sorting can be used instead [3, 8, 11, 13, 16]. However, the existing parallel sorting algorithms that have good worst-case time bounds are not easily implemented, and in some cases the hidden constants in the asymptotic running times are enormous [2]. Cole [9] shows how sorting-based parametric search can be optimized even further, but the optimization comes at the expense of making the technique even more complicated than it already is.

In this paper we show that Quicksort can be used as the generic algorithm in sorting-based parametric search, instead of a parallel sorting algorithm—see Section 3. Cole's optimization cannot be applied in this case, but in Section 4 we demonstrate that under certain assumptions that seem reasonable in practice, Cole's optimization is not needed to attain an efficient algorithm. These two observations considerably simplify the practical application of sorting-based parametric search.

Nevertheless, implementing algorithms based on parametric search remains a challenging task if one has to do it completely from scratch. Therefore, we implemented an object-oriented framework for parametric search in C++, that takes care of the most difficult parts of the technique. Using this framework, implementing algorithms that use parametric search becomes substantially easier, and for some applications it even becomes nearly trivial. The framework is very small—it consists of eight classes, four of

which are directly visible to the user. Each of these four classes has only a small number of member functions in its interface, and they are very easy to use. The framework is described in Section 5. Based on this parametric-search framework, we implemented Quicksort and Bitonic sort in such a way that they can be used as the generic algorithm in sorting-based applications of parametric search. These two sorting algorithms were then applied to two problems. The first one is the problem of finding the intersection of the median function of a set of monotone increasing linear functions with the *x*-axis. This problem was used by Megiddo to explain his technique [16], and has been used as a preliminary example in many other papers on parametric search. The second one is the Fréchet-distance problem [3]. We will give some details on these applications in Section 6. The experimental results presented in Section 7 indicate that the application of parametric search can indeed result in algorithms that are efficient in practice. In fact, while a simple binary-search approach is often advocated as a practical replacement for parametric search, it is outperformed by our Quicksort-based algorithm for the Fréchet distance problem.

2 Preliminaries

Megiddo's parametric-search technique [16] works as follows: assume that we have a decision problem $P(\lambda)$ that is monotonous in λ, i.e., if $P(\lambda_0)$ is true, then $P(\lambda)$ is true for all $\lambda < \lambda_0$. Our task is to find λ^*, the maximum value of λ for which $P(\lambda)$ is true. Suppose that we have an algorithm A_s that solves the decision problem P, and that can determine for any input value λ whether $\lambda < \lambda^*$, $\lambda = \lambda^*$, or $\lambda > \lambda^*$. Also suppose that the flow of control of A_s depends on comparisons, each of which depends on the sign of a polynomial in λ. Megiddo's idea is to run A_s generically on the unknown input λ^*. It may seem strange to run an algorithm on unknown input, while the outcome (namely, the verification that the input is equal to λ^*) is already known. However, we get to know the actual value of λ^* as a by-product of running A_s on λ^*. During the generic execution of A_s we maintain an open interval I in which λ^* is known to lie; initially, I is $(-\infty, \infty)$. Whenever a comparison has to be resolved, we need to determine the sign of a polynomial p at the value λ^*. This can be done without knowing the value of λ^*, by running the concrete version of A_s on the roots of p. This determines the location of λ^* among the roots, i.e., it either gives us two consecutive roots r_i and r_{i+1} such that $r_i < \lambda^* < r_{i+1}$, or we find out that one of the roots is λ^*. In the latter case we are done and we abort the execution of the generic algorithm. Otherwise, since the sign of a polynomial doesn't change in between two consecutive roots, we can determine the sign of $p(\lambda^*)$ by evaluating the polynomial p for any value $x \in (r_i, r_{i+1})$. This determines the outcome of the comparison, and, after updating I to $I \cap (r_i, r_{i+1})$, we resume the generic execution of A_s. During the execution of A_s, I gets progressively smaller, and we either run A_s to completion, ending up with a final interval I, or we find λ^* prematurely and abort the execution of A_s. In many applications of parametric search λ^* is one of the roots associated with the comparisons, and A_s will never run to completion in these cases. In other cases, such as in the "median-of-lines" example that Megiddo used to explain his technique [16], the structure of the problem restricted to the final interval I is simple, and the value of λ^* can be easily computed, given I.

If we denote the running time of A_s with T_s and the number of comparisons made by A_s with C_s, then the cost of running parametric search as described above is $O(C_s T_s)$. This can be improved by computing $A_s(r)$ only when we really have to. Recall that $P(\lambda)$ is monotonous: if $P(\lambda_0)$ is true, then $P(\lambda)$ is true for all $\lambda < \lambda_0$. Symmetrically, if $P(\lambda_0)$ is false, then $P(\lambda)$ is false for all $\lambda > \lambda_0$. This means that if we can somehow batch k comparisons, we can resolve the associated roots as follows: we find the median λ_m of the roots using a linear-time median-finding algorithm, and compute $A_s(\lambda_m)$. This determines the outcome of the decision problem for half of the roots, and we recursively deal with the remaining roots. It follows that if the number of roots associated with a comparison is bounded by a constant, we can resolve the k batched comparisons in a binary-search fashion with only $O(\log k)$ calls to A_s. Note that to be able to batch comparisons, they should be *independent*: the outcome of one comparison in a batch should not depend on the outcome of another comparison in the same batch. Megiddo [16] therefore suggests to replace the generic algorithm with a parallel version A_p, since the operations in one parallel step are usually independent. If A_p uses P processors and runs in T_p parallel steps, and we use the binary-search approach to resolve the comparisons in each parallel step, then the total cost of parametric search is $O(PT_p + T_p T_s \log P)$. Usually, the running time is dominated by the second term. Note that we don't actually run A_p on a parallel architecture; the parallelism is simulated and irrelevant in itself. The important point is the ability to collect

(preferably large) batches of comparisons in order to reduce the number of calls to the concrete decision algorithm A_s. Therefore, a weak model of parallelism, such as the parallel comparison model of Valiant, suffices. In this model, the complexity of an algorithm is only determined by the comparisons being made, and things like communication and synchronization between processes are ignored.

Cole [9] shows that in some applications of parametric search, the number of calls to the decision process can be reduced by a log-factor, thus improving the running time to $O(PT_p + T_pT_s + T_s \log P)$. We will explain his idea and comment on it in Section 4.

One of the drawbacks of parametric search mentioned by Agarwal and Sharir [1] is that it requires the design of an efficient parallel algorithm for the generic version of the decision problem, which is not always easy. However, it is instructive to point out that the generic algorithm does not necessarily have to solve the same problem as the concrete version; all that is required is that the output of the generic algorithm changes combinatorially at λ^*. In quite some cases, sorting can play the role of the generic algorithm; see for instance the spanning tree and scheduling problems studied by Megiddo [16], the slope-selection problem [11], the Fréchet-distance problem [3], the bottleneck-distance problem [13], and the problem of finding the minimum Hausdorff distance, under rotation and rotation, between two sets of points, lines, or polygons [8]. In several other cases, the generic algorithm consists of several steps, sorting being one of them [17, 19]. For sorting-based parametric search we have several parallel sorting algorithms at our disposal. The first sorting algorithm that sorts in $O(\log n)$ parallel steps using $O(n)$ processors is the sorting network of Ajtai et al. [2], usually referred to as the 'AKS-network'. When used for parametric search, it leads to a running time of $O(n \log n + T_s \log^2 n)$, or $O(n \log n + T_s \log n)$ when Cole's optimization [9] is applied. However, as many researchers have commented, this algorithm is rather complex and the constants hidden in the O-notation are very large. For practical use it is often suggested to use a sub-optimal parallel sorting algorithm instead, such as Valiant's merge sort [21], which runs in $O(\log n \log \log n)$ parallel steps using $O(n)$ processors. Less commonly known are the two sorting algorithms by Cole [10] that have the same time bounds as the AKS-network and also use $O(n)$ processors. One of these algorithms works under the CREW PRAM model of parallel computation, and the constants in the running time are small. However, it does not meet the conditions for applying Cole's optimization [9]. The other one works under the EREW PRAM model; it is more complex than the first algorithm, but Cole's optimization can be applied to it.

In fact, parallelism is not a requirement for resolving the roots in a binary-search fashion. It suffices to be able to collect the roots associated with independent comparisons in a small number of batches. In this paper we show that Quicksort, quite surprisingly, meets the requirements, which leads to a considerable simplification of sorting-based parametric search, yielding an expected running time of $O(n \log n + T_s \log^2 n)$ (see Section 3). Cole's optimization cannot be applied here, but we also show that under certain conditions that seem not too unlikely in practical situations, the expected running time of parametric search will be $O(PT_p + T_s \log(P + T_p))$, rather than $O(PT_p + T_pT_s \log P)$, even without Cole's optimization. For Quicksort-based parametric search this means that the expected running time is $O(n \log n + T_s \log n)$ if these conditions, which we will explain in Section 4, are met.

Parametric-search has always been more popular with theoreticians than with practitioners. While parametric search, when applied to the right problems, often leads to asymptotic running times that are about an order of magnitude better than the alternatives, it is often advised against using the technique in practice. The reason for this negative advice is usually twofold. Firstly, since parametric search is a complicated technique, it is believed that implementing it must also be complicated and hard. To our knowledge, the technique has been implemented only twice; the first time by Toledo [20], who applies it to the "median-of-lines"-problem that Megiddo gave as a simple illustration of parametric search [16], and to the "slope selection problem" [11]. The second implementation that we know of is by Schwerdt et al. [19], who implement the algorithm for finding the minimum diameter of a set of moving points by Gupta et al. [14]. The second reason why parametric search is hardly ever used in practice is that it is generally assumed that the overhead involved (i.e., the hidden constants in the O-notation) are too large to be of practical use. Partly because of the presumed difficulties with parametric search, alternatives have been proposed for many specific problems; see for instance Agarwal and Sharir [1] and Chan [7]. However, parametric search is currently still the most general method. We would like to counter-balance the advice of avoiding parametric search in real-world applications by making some observations that considerably simplify sorting-based parametric-search (Sections 3 and 4), and by providing a framework that takes care of much of the difficulties of parametric search, whether sorting-based or not (Section 5).

3 Quicksort-based parametric search

Until now, sorting-based parametric search has always relied on parallel sorting algorithms for the batching of comparisons. Megiddo [16] suggests the parallel sorting algorithms of Valiant [21] or Preparata [18]. Other authors propose using the AKS-network [2] to derive running times that are asymptotically faster. Cole [9] originally applies his optimization technique to the AKS-network. In a later paper [10] he gave two parallel sorting algorithms that have the same asymptotic time bounds as the AKS-network, but with much smaller constant factors hidden in the O-notation.

In this paper we show that Quicksort can be used as the generic algorithm for sorting-based parametric search. There is no need to use a parallel version of Quicksort; a serial version suffices to be able to batch comparisons. For a complete description and analysis of Quicksort, see any introductory textbook on algorithms, for instance the one by Cormen, Leiserson, and Rivest [12].

In short, Quicksort sorts an input array A with elements numbered $1 \ldots n$ as follows:

1. If A has less than two elements, return;

2. Otherwise, choose A[1] as the *pivot element*;

3. Partition A into two sub-arrays $A_<$ (containing the elements of A that are smaller than the pivot element) and $A_\geq$ (containing the elements of A that are greater than or equal to the pivot element);

4. Recursively sort $A_<$ and $A_\geq$.

Quicksort has a worst-case running time of $\Omega(n^2)$; in fact, if the input is (almost) sorted, the running time is indeed quadratic in the input size. To attain an $O(n \log n)$ expected running time, randomization is employed, either by randomly permuting the input, or by choosing a random element of the array as the pivot element in step 2. Quicksort is very simple, well-known, and in practice it usually outperforms sorting algorithms with a deterministic $O(n \log n)$ running time.

The third step of the algorithm is usually performed by maintaining two pointers. One of these initially points to the first element of the array, and moves toward the higher elements, until an element e_i is found that is greater than or equal to the pivot element. The other pointer initially points to the last element of the array, and moves toward the lower elements, until an element e_j is found that is less than the pivot element. If e_i comes before e_j in the array, they are swapped, and the process continues until the pointers pass each other.

Observe that in step three of the algorithm, all comparisons between elements of the array are independent: we compare all elements with the pivot element. This implies that we can use Quicksort in the parametric-search setting: we collect all the comparisons in step three, and resolve them in a binary-search fashion as Megiddo suggested [16]. After all comparisons in the third step have been resolved, we do the actual partitioning as described in the previous paragraph.

To make Quicksort-based parametric search efficient, we would like to collect the comparisons for all recursive calls at the same recursion level l in a single batch, rather than resolving them in $O(2^l)$ separate batches. This is trivial if we replace the recursion by iteration. We maintain a list of (pointers to) sub-arrays; initially, this list contains only one array, namely, the input array A. In each iterative step we first collect the $O(n)$ comparisons of all sub-arrays (i.e., for each sub-array we compare all elements of the sub-array with the pivot element of the sub-array), and resolve them in a binary-search fashion. Next, we partition each sub-array as described before; this doubles the size of the list of sub-arrays. From this list, we remove the sub-arrays of size 1, and we proceed with the next iteration.

If we employ randomization by either randomly permuting the input as a first step of the algorithm, or by choosing a random element of each sub-array as the pivot element, the expected number of iterations is $O(\log n)$. In each iteration we do $O(n)$ work in total, and the number of calls to the algorithm A_s that solves P is $O(\log n)$ in each iteration. It follows that when we use Quicksort as the generic algorithm in sorting-based parametric search, the running time will be $O(n \log n + T_s \log^2 n)$. It seems not possible to apply Cole's optimization [9]—we cannot partition a sub-array before all comparisons for that sub-array have been resolved—and therefore we cannot reduce the running time to $O(n \log n + T_s \log n)$. However, in the next section we will show that under certain conditions that seem reasonable in practice, the running time will be closer to $O(n \log n + T_s \log n)$ than to $O(n \log n + T_s \log^2 n)$. Evidently, using Quicksort as the

generic algorithm for sorting-based parametric search considerably simplifies the implementation. Also, since Quicksort has proved to be very efficient in practice, it can be expected to lead to faster algorithms than when complicated parallel algorithms with larger constant factors in the running time are used.

Note that batching of comparisons is not a prerequisite of parametric search. Without batching, Quicksort and other serial sorting algorithms can also be used as the generic algorithm of sorting-based parametric search, at the cost of a (much) higher running time. In fact, Toledo[20] gives an implementation of a non-batched Quicksort-based parametric-search algorithm, apart from a version that uses Valiant's parallel merge sort[21]. However, our observation that Quicksort can be used to batch comparisons makes implementing optimized versions of parametric search considerably easier.

4   Cole's optimization revisited

Cole [9] shows that in some cases the running time of parametric search can be reduced from $O(PT_p + T_pT_s\log P)$ to $O(PT_p + T_pT_s + T_s\log P)$. His optimization technique works as follows: instead of resolving a batch of $O(P)$ roots by $O(\log P)$ calls to A_s in a binary-search fashion (which requires that the roots are sorted first), we find the median root r (which can be done in linear time) and call $A_s(r)$. This determines the outcome of the decision problem for half of the roots. Assume for the moment that the comparisons made by the generic algorithm each depend on only one root. Then after this single call to A_s, the outcome of half of the comparisons that depend on the roots in this batch is known. The (parallel) processes that depend on these comparisons can then be resumed, which will lead to new comparisons and associated roots. These roots are then joined with the first batch (of which only half the number of roots have been resolved so far), and the process repeats with finding the median root, and so on. Cole shows that if the roots are weighted according to some clever scheme and a weighted median finding algorithm is used, the number of calls to A_s will be reduced by a log-factor. The technique still works if the comparisons depend on a constant number of roots, rather than on a single root. The drawbacks of the technique are that it cannot generally be applied, as it imposes some conditions on the parallel algorithm, and it also makes the generic algorithm more complicated.

In our experiments (see Section 7) we did not apply Cole's optimization, but we resolved each batch of roots in $O(\log P)$ steps, as in Megiddo's original paper [16]. We noticed that the number of calls to the decision process was indeed growing with the input size, yet much less than we expected. We attributed this to the maintenance of the progressively smaller interval in which λ^* is known to lie. Recall that resolving a batch of roots either gives us the value of λ^* (namely, if λ^* is one of the roots), or we end up with an interval (r_k, r_l), where r_k and r_l are roots from one of the batches processed so far, and none of the other roots from these batches lie in-between r_k and r_l. When we resolve the next batch of roots, we can disregard the roots that do not lie in this interval, as the outcome of the decision process for these roots is already known: $P(r) = $ true if $r \leq r_k$, and $P(r) = $ false if $r \geq r_l$. Hence, we only have to do the binary search for roots $r \in (r_k, r_l)$. Intuitively, since this interval gets smaller in each roots-resolving step, we will have to consider fewer and fewer roots as the algorithm progresses.

Let us assume for the moment that the roots are uniformly distributed over the batches. Number the batches that are processed during the algorithm from 0 to m. In Megiddo's setting, $m = O(T_p)$, where T_p is the number of parallel steps of the generic algorithm. In step 0, the interval that gives the bounds on λ^* is $(-\infty, \infty)$, and we call A_s $O(\log P)$ times to resolve the decision process for all roots in a binary-search fashion. At the beginning of step i, for $1 \leq i \leq m$, the outcome of $P(r)$ for all roots r from the batches solved in steps 0 to $i-1$ is known; the total number of these roots is $i \cdot P$. Furthermore, the interval that we maintain is bound by two consecutive roots r_k and r_l from this set of $i \cdot P$ roots. Since the number of roots for which we have to call A_s in step i is P, the expected number of roots in step i that lie inside (r_k, r_l) is $1/i$ (under our assumption that the roots are uniformly distributed over the $k+1$ batches). Even if we call A_s for all the roots that lie in (r_k, r_l) instead of doing a binary search, the expected total number of calls to A_s, summed over steps 1 to m, is $\sum_{i=1}^{m} 1/i$, which is $O(\log m)$. It follows that the expected running time of parametric search is $O(PT_p + T_s\log(P + T_p))$. Since T_p (the number of parallel steps of the generic algorithm) is usually smaller than P (the number of processors), this is $O(PT_p + T_s\log P)$. With Quicksort, we get an expected running time of $O(n\log n + T_s\log n)$.

Of course, we cannot generally assume that the roots are uniformly distributed over the batches; it

may be the case that in each step i, for $1 \leq i \leq m$, all the roots in that step lie inside the interval that results from the previous steps, which means that we have to do the binary search over all these roots. However, it seems to us that in many practical situations this extreme situation is unlikely to occur, and we expect that in practice the distribution of the roots over the batches is "sufficiently uniform" to benefit from maintaining the bounds on λ^*; If we use Quicksort as the generic algorithm, randomly permuting the input as a preprocessing step may even help in this respect, although this still doesn't give any guarantees. However, our experiments with the Quicksort-based parametric-search solution to the Fréchet-distance problem show that the bulk of the invocations of the decision algorithm is indeed done in the first 10%–30% of the iterations (see Section 7).

5 The parametric-search framework

Batching of comparisons isn't always possible or necessary in applications of parametric search, but when it is applicable it usually reduces the number of calls to the decision process by about an order of magnitude. At the same time, it also makes implementing algorithms based on parametric search difficult, because it disrupts the normal flow of control. Suppose for example that we have a function `f()` (in pseudo-C++ code) as in Figure 1.

```
1.  void f(){
2.      int i, j, k;
3.
4.      g1(i); g2(j); g3(k);   // call by ref.
5.      i += j+k;
6.      h1(i); h2(i); h3(i);
7.  }
```

Figure 1: Original code.

Assume that we want to execute this code in the parametric-search setting, and that the comparisons made by the functions `g1()`, `g2()`, and `g3()` are mutually independent. Naturally, we would like to batch them. Similarly, assume that the comparisons made by `h1()`, `h2()`, and `h3()` are independent and we also want to batch those. However, the latter three functions depend on the variable `i`, which in turn depends on the results of the three functions in line 4, so we cannot collect and resolve the comparisons of the six functions in lines 4 and 6 in a single batch. What we have to do instead is starting the execution of `g1()`, collect its comparison and suspend its execution, do the same for `g2()` and `g3()`, resolve the comparisons collected so far, and resume the execution of the three functions. Next, we can execute the statement in line 5, and we get another round of starting functions, collecting comparisons and suspending execution, resolving comparisons, and resuming execution, this time of the functions in line 6. It gets even more cluttered when we want to simulate parallelism, i.e., when the order of execution of the functions in line 4 can be arbitrary, as well as the order of execution of the functions in line 6.

Our parametric-search framework takes care of starting, suspending and resuming of functions, and of collecting and resolving comparisons. The framework consists of a small hierarchy of C++ classes. A user needs to interact with only four of these classes, and the number of functions in the public interface of these classes is very small. These four classes are named `Scheduler`, `Process_base`, `Comparison_base`, and `Root`, respectively. The idea is that functions that need to be started, suspended, and resumed are turned into objects of a (user-defined) class derived from `Process_base`. We will call such objects *processes*. (Note, however, that they are unrelated to processes in operating systems; instead, the are regular C++ objects.) Similarly, comparisons are to be implemented by the user by deriving from `Comparison_base`. Let us give an example by transforming the code above into such a collection of classes; see Figure 2. The code is incomplete and simplified, but shows the important principles.

The transformation is straightforward; functions that perform independent comparisons (such as the three functions in line 4 in Figure 1) correspond to dynamically created objects; the creation of independent objects is done by a single member function (lines 3–7 in Figure 2). Similarly, the code in lines 5 and 6 in

```
1.  class proc_f : public Process_base {
2.  public:
3.    void memfun_1(){
4.       spawn( new proc_g1(i) );
5.       spawn( new proc_g2(j) );
6.       spawn( new proc_g3(k) );
7.    }
8.
9.    void memfun_2(){
10.      i += j+k;
11.      spawn( new proc_h1(i) );
12.      spawn( new proc_h2(i) );
13.      spawn( new proc_h3(i) );
14.    }
15.
16.  private: // member variables
17.     int i, j, k;
18.  };
```

Figure 2: Transformed code.

Figure 1 is translated into lines 9–14 in Figure 2. Finally, local variables in the original code correspond to member variables in the transformed code. The two member functions memfun_1() and memfun_2() are registered with the framework in the constructor of proc_f (not shown here). Processes themselves, such as proc_f and the processes it creates, are also registered with the framework; this is done automatically. The spawn function takes care of linking 'child processes' to their 'parent process'.

When processes are created, they are not executed immediately; the Scheduler decides when a process should be started. When it is time for proc_f in Figure 2, the Scheduler tells it to execute its first registered member function, memfun_1(). As a result, three new processes are created (but not executed immediately). After memfun_1() has finished, proc_f is suspended. The Scheduler then selects other processes to run, including the child processes of proc_f. When all child processes spawned by proc_f have finished, it is commanded by the Scheduler to resume its execution, starting with the next registered member function, memfun_2(), and so on. When there are no more member functions, proc_f is deleted by the Scheduler.

Processes can spawn other processes and comparisons (objects of a user-defined class derived from Comparison_base). Comparisons in turn can spawn objects of class Root. The Scheduler takes care of resolving the roots in a binary-search fashion, of signaling the comparisons that all their associated roots have been resolved so that the outcome of the comparison can be determined, and of starting, suspending, resuming, and terminating processes and comparisons. Details can be found in the reference manual and tutorial that come with the software (see the CGAL extension package section on http://www.cgal.org); what we hope to show here by means of the simple code examples and the brief explanation is the simplicity of the framework.

The user needs no knowledge of the inner workings of the framework, but only has to do a simple transformation similar to the one from the code shown in Figure 1 into the code shown in Figure 2. Naturally, more complicated constructs can also be translated. For instance, iteration can be expressed by having member functions register themselves again with the framework after they have finished their task, and recursion is expressed by having a process spawn another process of its own class.

The generic algorithm that is run on λ^* (the unknown solution to the optimization problem) can in certain cases be replaced by sorting. To simplify the implementation of parametric search even further in these cases, we provide implementations of two sorting algorithms. The first one is Bitonic sort, the parallel sorting network by Batcher [4], which uses $O(n)$ processors to sort n input items in $O(\log^2 n)$ parallel steps. Our implementation of Bitonic sort uses the parametric-search framework to simulate the

parallelism and to resolve comparisons in batches. The second sorting algorithm is Quicksort; here, we also use the framework to collect and resolve the comparisons in the partitioning step, prior to doing the actual partitioning (see Section 3).

Using the framework to do sorting-based parametric search requires the user to implement the following classes or functions:

1. A class or function that computes the items that are to be sorted from the input. The input may consist of these items themselves;

2. A class derived from `Comparison_base` that computes the roots associated with the comparison of two items, and that can determine the outcome of the comparison once the solution of the decision problem P is known for all these roots;

3. A class or function that solves the decision problem.

Furthermore, the user should specify whether to use Bitonic Sort or Quicksort, and decide on the number type to be used for the calculations.

The parametric-search framework and the two sorting algorithms have the same design philosophy as CGAL [5], the library of algorithms for computational geometry. Our framework doesn't depend on CGAL, but the two cooperate very well. We consider this important, since it is our goal to use the framework for implementing algorithms that solve geometric optimization problems. In fact, we plan to make our framework available as a CGAL extension package.

6   Applications

Using our framework and the two sorting algorithms, we implemented two algorithms that apply sorting-based parametric-search. The first one solves the 'median-of-lines' problem that Megiddo gave as a simple illustration of parametric search [16]. Our motivation for implementing this algorithm is that it gives a simple means of testing our framework, and at the same time it enables us to easily explain the framework in a tutorial. The second algorithm that we implemented is the one by Alt and Godau [3] for computing the Fréchet distance between two polygonal curves. We will discuss it here briefly to illustrate how the framework is used.

The usual informal illustration of the Fréchet metric is the following: suppose a man is walking his dog, keeping it on a leash. The man is walking on a polygonal curve P, and the dog on a polygonal curve Q. Both are allowed to control their speed, which may have any non-negative value (i.e., they are allowed to stop, but they cannot go back). Then the Fréchet distance between P and Q is the minimal length of the leash that is necessary.

The two following definitions from Alt and Godau [3] characterize the Fréchet metric more formally; in these definitions, V denotes an arbitrary Euclidean vector space.

Definition 1 *A* curve *is a continuous mapping* $f : [a,b] \to V$ *with* $a,b \in \mathbb{R}$ *and* $a < b$. *A* polygonal curve *of length n is a curve* $P : [0,n] \to V$ *with* $n \in \mathbf{N}$, *such that for all* $i \in \{0,1,\ldots,n-1\}$ *each* $P_{|[i,i+1]|}$ *is affine, i.e.,* $P(i+\lambda) = (1-\lambda)P(i) + \lambda P(i+1)$ *for all* $\lambda \in [0,1]$.

Definition 2 *Let* $f : [a,a'] \to V$ *and* $g : [b,b'] \to V$ *be curves. Then* $\delta_F(f,g)$ *denotes their* Fréchet distance, *defined as*

$$\delta_F(f,g) := \inf_{\substack{\alpha:[0,1]\to[a,a'] \\ \beta:[0,1]\to[b,b']}} \max_{t \in [0,1]} ||f(\alpha(t)) - g(\beta(t))||$$

where α, β *range over the continuous and increasing functions with* $\alpha(0) = a$, $\alpha(1) = a'$, $\beta(0) = b$, *and* $\beta(1) = b'$.

Let P and Q be polygonal curves, with m and n edges, respectively. Alt and Godau first show how to solve the case $m = n = 1$. Here, P and Q are simple line segments.

Define $F_\lambda := \{(s,t) \in [0,1]^2 \mid d(P(s),Q(t)) \leq \lambda\}$. F_λ is called the "free space"; it is shown in Figure 3, cited from Alt and Godau [3], together with P, Q, and λ. It is proven that if the distance measure between

two points on either polygonal curve is Euclidean, then the intersection of F_λ with the unit square is an ellipse.
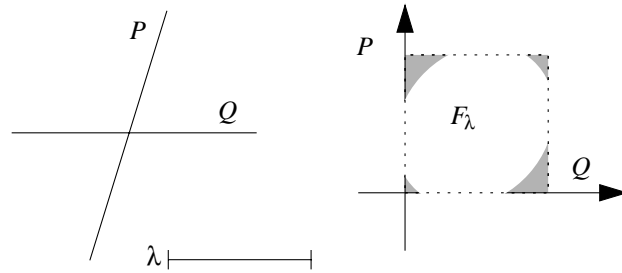

Figure 3: P, Q, λ, and F_λ.

The definition of F_λ is extended to arbitrary polygonal curves P and Q with m and n edges, respectively: $F_\lambda := \{(s,t) \in [0,m] \times [0,n] \mid d(P(s),Q(t)) \leq \lambda\}$. Figure 4, also cited from Alt and Godau, shows an example for two polygonal curves with 6 and 4 segments.
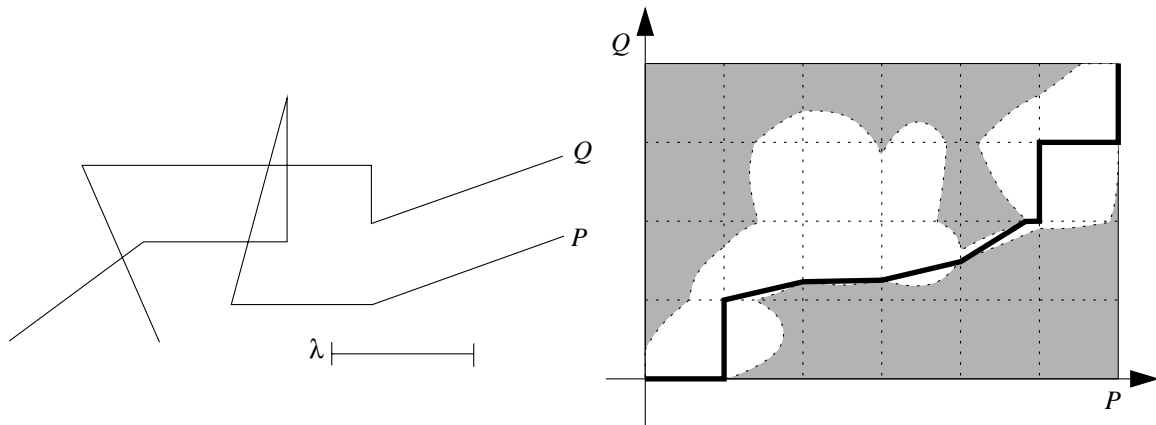

Figure 4: Diagram for polygonal chains P, Q and the given λ.

The algorithm for the decision problem is based on the observation than given two polygonal curves P and Q, $\delta_F(P,Q) \leq \lambda$ exactly if there is a curve within the corresponding F_λ from $(0,0)$ to (m,n) that is monotone in both directions.

The critical values in the decision process are the coordinates of the intersections of the ellipses with the boundaries of the squares in the diagram (see Figure 4). Since the number of intersections is a constant for each square, there are $O(mn)$ critical values. They depend on λ, the input parameter of the decision problem; in fact, they are polynomials p_i in λ, where the index i ranges over the $O(mn)$ possibilities. Alt and Godau show that for $\lambda = \lambda^*$, either two critical values in a row or two critical values in a column of the diagram have the same value (there are some other cases that need to be considered, but we ignore them here for simplicity). This means that sorting-based parametric search can be used to find the Fréchet distance, because if $p_i(\lambda^*) = p_j(\lambda^*)$, then p_i and p_j will end up as adjacent polynomials after sorting. Therefore, the sorting algorithm has to compare p_i and p_j (otherwise the order of p_i and p_j cannot be known), and this comparison involves the computation of the roots of $p_i - p_j$; one of these roots is λ^*.

The first C++ class that we implement solves the decision problem. This takes some effort, but it isn't overly complicated. For a given λ we compute the intersections of the ellipses with the boundaries of the squares in the diagram. In a subsequent step, we determine if there is a path from $(0,0)$ to (m,n) within F_λ that is monotone in both directions. See the paper by Alt and Godau [3] for details.

The items that are to be sorted are the intersections of ellipses with the boundaries of the squares in the diagram for P and Q; these are polynomials in the input parameter λ of the decision process. The second class that we implement computes these $O(mn)$ polynomials from P and Q, and we tell the framework that it should use this class to retrieve the input for the sorting algorithm.

Finally, the third class that we implement compares two polynomials; it is derived from `Comparison_base`. Two polynomials p_i and p_j are compared by computing the roots of $p_i - p_j$. These roots are collected by the framework and resolved in a binary-search fashion. Once the outcome of the decision process for all roots of $p_i - p_j$ is known, the framework tells the comparison class to determine the outcome of the comparison of p_i and p_j. This is done by evaluating p_i and p_j for an arbitrary value in the interval in which λ^* is known to lie; this interval is guaranteed not to contain any of the roots of $p_i - p_j$ at the time the framework tells the comparison class to determine the outcome of the comparison.

These three classes are all we need to implement to solve the optimization problem (i.e., to compute the Fréchet distance between P and Q). The test results of this implementation are presented in the next section.

The decision problem considers $O(mn)$ critical values and can be solved in $O(mn)$ time. If we use Quicksort as the generic sorting algorithm, then the expected number of calls to the decision process (without any assumptions on the distribution of the roots) is $O(\log^2(mn))$. The expected overhead of the sorting itself is $O(mn\log(mn))$, so the total expected running time is $O(mn\log^2(mn))$. However, if the distribution of the roots over the subsequent steps of the algorithm is sufficiently random, we may expect running times that are closer to $O(mn\log(mn))$, as we showed in Section 4.

7 Experimental results

We tested the algorithm for computing the Fréchet distance between two polygonal curves of n vertices each, for $n \in \{16, 32, 64, 128\}$. The polygonal curves were created by generating $2n$ random points in the unit square. We computed the Fréchet distance by running the sorting-based parametric search algorithm described in the previous section; both Bitonic sort and Quicksort were used as the generic sorting algorithm. The first number type that we used was `long double`; this means that we didn't get an exact answer, but only an approximation.

We also computed an approximation by doing a binary search on the values representable by a `long double`; this is often suggested as an alternative for parametric search. Since a `long double` has 96 bits on our system, we expected that the number of calls to the decision process would also be 96. In reality, we found a much higher number of iterations. This is explained by the fact that the distance between two consecutive numbers representable by a `long double` is much smaller around 0 than it is around large positive and negative values. In fact, it is possible to do a binary search on the bits of a `long double`, but rather than implementing such an efficient binary search, we simply computed the running time for 96 iterations from the actual running time and number of iterations. Such a computation is possible, since the overhead in both methods of binary search is negligible; almost all time is spent in the decision process.

All three algorithms were run on the same (randomly generated) input, and for each input size of $n \in \{16, 32, 64, 128\}$ we did 100 repetitions. The results are presented in Table 1; the running times for binary search are normalized for 96 iterations as described above. The experiments were done on a PC with a 667 MHz Pentium III processor and with 128 Mb memory, running Linux.

In nearly all cases, Quicksort was faster than the two other methods. In one of the repetitions for input size 32 however, the running time was very high: 0.511 seconds, compared to 0.088 seconds for the second-highest running time. This may be due to the fact that Quicksort has an expected-case performance of $O(n\log n)$, but a worst-case performance of $O(n^2)$.

Both for Bitonic sort and Quicksort, the framework aborts the sorting when λ^* is found as the root. The number of times this actually happens is listed in the table. In the binary search method we also stop when we encounter λ^*; however, difference between the smallest and the highest number of iterations was less than 2 percent.

We also tested the Fréchet-distance algorithms using `leda_real` [6]. This number type provides exact relational operators $(=, \neq, <, \leq, >, \geq)$ and is closed under addition, subtraction, multiplication, division, and computation of k-th roots. Exact comparisons with `leda_real` are generally efficient, but at times they

can be very expensive, especially when two complicated expressions are compared that are actually equal. In our first trials with polygonal curves of less than 10 vertices each, the running times were measured in hours. Simplifying the arithmetic expressions in the decision process reduced the running times to a few minutes for polygonal curves of 16 vertices each. We still consider this too slow, but we see a lot of possibilities for optimizations. Specifically, by maintaining some extra information in the data structure for the decision algorithm, we can avoid the comparison of expressions that we know to be equal a priori—and these are the comparisons that are most expensive.

To verify the assumption that the maintenance of the progressively smaller interval of bounds on λ^* results in fewer and fewer invocations of the decision algorithm as the generic algorithm progresses (Section 4), we recorded the number of calls in each iteration of the Quicksort-based Fréchet-distance algorithm on randomly generated input of size $n = 100$. It turned out that for the cases without early abort, all invocations of the decision algorithm were done in the first 10%–30% of the iterations (recall that we implemented an iterative rather than a recursive version of Quicksort; see Section 3).

Input size	Method	min T	avg T	max T	# times aborted
$16 + 16$	Bitonic sort	0.005	0.019	0.023	26
	Quicksort	0.004	0.010	0.012	24
	Binary search	0.013	0.014	0.016	—
$32 + 32$	Bitonic sort	0.030	0.172	0.247	29
	Quicksort	0.027	0.073	0.511	29
	Binary search	0.061	0.072	0.084	—
$64 + 64$	Bitonic sort	0.133	1.239	1.492	25
	Quicksort	0.122	0.409	0.518	26
	Binary search	0.671	0.685	0.693	—
$128 + 128$	Bitonic sort	0.663	6.042	7.469	29
	Quicksort	0.729	1.864	2.407	29
	Binary search	2.704	2.769	2.810	—

Table 1: Test results for the Fréchet-distance algorithms. (Times are in seconds.)

8 Concluding remarks

In this paper, we observed that Quicksort may be used to efficiently implement algorithms that use sorting-based parametric search. We also argued that Cole's optimization [9] may not be needed under certain conditions that seem reasonable in practice. We presented a framework that allows users to implement various applications of parametric search in a simple and efficient way. The two sorting algorithms that we provide with the framework can be used as the generic algorithm in sorting-based parametric search. Especially Quicksort performs very well; our Quicksort-based implementation of the algorithm by Alt and Godau [3] gives better result than the simple binary-search approach.

References

[1] P. K. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *ACM Comput. Surv.*, 30:412–458, 1998.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[3] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Internat. J. Comput. Geom. Appl.*, 5:75–91, 1995.

[4] K. E. Batcher. Sorting networks and their applications. In *Proc. 1968 Spring Joint Computer Conf.*, pages 307–314, Reston, VA, 1968. AFIPS Press.

[5] H. Brönnimann, L. Kettner, S. Schirra, and R. C. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. In *Generic Programming — International Seminar on Generic Programming*, volume 1766 of *Lecture Notes Comput. Sci.*, pages 206–217. Springer-Verlag, 2000.

[6] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, January 1996.

[7] T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete Comput. Geom.*, 22(4):547–567, 1999.

[8] L. P. Chew, M. T. Goodrich, D. P. Huttenlocher, K. Kedem, J. M. Kleinberg, and D. Kravets. Geometric pattern matching under Euclidean motion. *Comput. Geom. Theory Appl.*, 7:113–124, 1997.

[9] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987.

[10] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[11] R. Cole, J. Salowe, W. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Comput.*, 18(4):792–810, 1989.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[13] A. Efrat and A. Itai. Improvements on bottleneck matching and related problems using geometry. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 301–310, 1996.

[14] P. Gupta, R. Janardan, and M. Smid. Fast algorithms for collision and proximity problems involving moving geometric objects. *Comput. Geom. Theory Appl.*, 6:371–391, 1996.

[15] N. Megiddo. Combinatorial optimization with rational objective functions. *Math. Oper. Res.*, 4:414–424, 1979.

[16] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.

[17] S. Nakano, T. Nishizeki, T. Tokuyama, and S. Watanabe. Labeling points with rectangles of various shapes. In *Graph Drawing, 8th International Symposium (GD 2000)*, volume 1984 of *Lecture Notes Comput. Sci.*, pages 91–102. Springer-Verlag, 2001.

[18] F.P. Preparata. New parallel-sorting schemes. *IEEE Trans. Comput.*, C-27:669–673, 1978.

[19] J. Schwerdt, M. Smid, and S. Schirra. Computing the minimum diameter for moving points: an exact implementation using parametric search. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 466–468, 1997.

[20] S. Toledo. Extremal polygon containment problems and other issues in parametric searching. M.s. thesis, Dept. Comput. Sci., Tel Aviv Univ., Tel Aviv, 1991.

[21] L. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4(3):348–355, 1975.