Parametric Search Using CGAL*

(abstract)

René van Oostrum† Remco C. Veltkamp†

1 Introduction

Parametric search, the optimization technique by Megiddo [2], computes a value λ^* that optimizes an objective function f. The objective function corresponds with a monotonic decision problem $\mathcal{P}$: if $\mathcal{P}(\lambda_0)$ is true, then $\mathcal{P}(\lambda)$ is true for all $\lambda < \lambda_0$. Parametric search solves the optimization problem with the use of an algorithm $\mathcal{A}_s$ that solves the corresponding decision problem $\mathcal{P}$. The idea is to run a "generic" version of the decision algorithm on the unknown value λ^*. This generic algorithm uses the concrete version of $\mathcal{A}_s$ to determine the outcome of the decision problem for a set of concrete values; for one of these values λ, $\mathcal{A}_s$ will report that $\lambda = \lambda^*$.

The ¤ow of control of the generic algorithm depends on the outcome of comparisons, each of which in turn depends on the sign of a polynomial at the value λ^*. Evaluating these polynomials is usually expensive. Megiddo shows how to improve running times by about an order of magnitude: instead of evaluating the comparisons immediately, they are batched. The algorithm for the decision problem is applied to the roots of the polynomials associated with the comparisons; these are the *critical values* of the algorithm. It is not necessary to test *all* critical values, since the decision problem is monotonic. This means that we can resolve the batch of comparisons in a binary-search fashion. The comparisons that are resolved in a single batch must be independent: the outcome of one comparison should neither depend on nor in¤uence the outcome of another comparison in the same batch. Therefore, Megiddo suggests to replace the generic algorithm $\mathcal{A}_s$ by a parallel version $\mathcal{A}_p$, as parallel algorithms naturally tend to partition the work into independent parts that can be computed simultaneously. For more details, we refer to Megiddo's paper [2] and to our SoCG'02 paper [3].

Parametric search has always been considered to be mainly of theoretical interest; many researchers remark that the technique is very dif£cult to implement, and that the hidden constants in the "big-oh"-notation for the running time are too large for practical use. It is often suggested to replace parametric search with a binary search over the space of representable numbers. However, this is not possible if one uses a number type for exact arithmetic, such as `leda_real`.

We implemented a C++ framework that takes care of the dif£cult parts of the parametric search technique: suspending and resuming of computation, resolving batches of comparisons, and synchronization of several kinds of interdependent computational tasks. The framework greatly reduces the complexity involved with the application and implementation of parametric search. It is small, easy to use, and surprisingly ef£cient. The framework will be made available as a CGAL extension package.

In our SoCG'02 paper [3] we describe *what* the framework does, and we give the theoretical background on which the framework is based. In our talk in the CGAL User Workshop (and in this abstract) we will give an overview of *how* the framework works.

*This research was supported by the Dutch Technology Foundation STW, project UIF 5055 (SHAME: Shape Matching Environment.)

†Institute of Information & Computing Sciences, Utrecht University, P.O.Box 80.089, 3508TB, Utrecht, The Netherlands. Email: {rene,Remco.Veltkamp}@cs.uu.nl

2 Overview of the framework

The parametric-search framework consists of a small number of classes, four of which are visible to the user: `Scheduler`, `Process_base`, `Comparison_base`, and `Root`.

The user must de£ne one or more so-called *process classes* by deriving from class `Process_base`, and one or more *comparison classes* by deriving from `Comparison_base`. *Processes* (objects of a process class) may instantiate other processes and *comparisons* (objects of a comparison class). The roots of the polynomials associated with the comparisons are represented by objects of class `Root`.

Since comparisons are not evaluated immediately, and the ¤ow of control of a process may depend on the outcome of comparisons or on other processes, it should be possible for a process to suspend its execution, and to resume when the outcome of the comparisons or processes on which it depends is known. The necessary machinery is provided in the four aforementioned classes of the framework. A user must derive her process classes from `Process_base`, and partition the computation of the process classes in one or more member functions. These member functions should be scheduled using a member function of `Process_base`. The `Scheduler` then calls these scheduled member functions when it is time for a process to perform the next step of its computation.

When a process or comparison has £nished its computation, it is deleted by the framework. The destructor of `Process_base` or `Comparison_base`, respectively, noti£es the parent process (i.e., the process that created the process or comparison being deleted). When all processes and comparisons created by a process have been deleted, it is time for the process to resume its execution, and it is told to do so by the `Scheduler`, as explained above.

Similarly, the outcome of a comparison can only be evaluated when the outcome of the decision algorithm is known for all its associated roots. The `Scheduler` runs the decision algorithm in a binary search fashion on the roots. Roots for which the outcome is known are deleted by the `Scheduler`, and upon destruction they notify the comparison with which they are associated. When all roots of a comparison have been deleted, the `Scheduler` tells the comparison to evaluate itself (by calling a pure virtual function of `Comparison_base`, to be overloaded by the comparison classes).

The relations between the four main classes of the framework are depicted in Figure 1.

3 Sorting-based parametric search

In several cases, the generic algorithm that drives the parametric search can be replaced by sorting the polynomials associated with the comparisons. As long as the sorting algorithm compares at least the same polynomials as the generic algorithm it replaces, we are guaranteed to £nd the optimal value λ^*. When sorting can be applied, it is often an attractive alternative, since ef£cient sorting algorithms are readily available, while designing an ef£cient parallel algorithm for the problem at hand may be dif£cult.

However, implementing a sorting algorithm in such a way that it is suitable for parametric search is dif£cult if one has to do it from scratch. One still has to batch comparisons and resolve them in a binary-search fashion, and suspending and resuming of execution is still needed. But since the implementation of such sorting algorithms would be an interesting test case for our framework, and the results could be reused in any application of parametric search for which the generic algorithm can be replaced by sorting, we decided to implement two sorting algorithms using our framework. We chose Bitonic sort and Quicksort (for a discussion on why one would use Quicksort for parametric search, instead of a parallel sorting algorithm, we refer to our SoCG'02 paper [3]).

4 Experiments

We implemented the algorithm by Alt and Godau [1] for computing the Fréchet distance between two polygonal curves. This algorithm uses sorting-based parametric search, and we employed both our Bitonic Sort implementation and our Quicksort implementation. The results (summarized in our SoCG'02 paper [3]) are promising; the Quicksort version outperforms a simple binary search of the space of representable numbers.

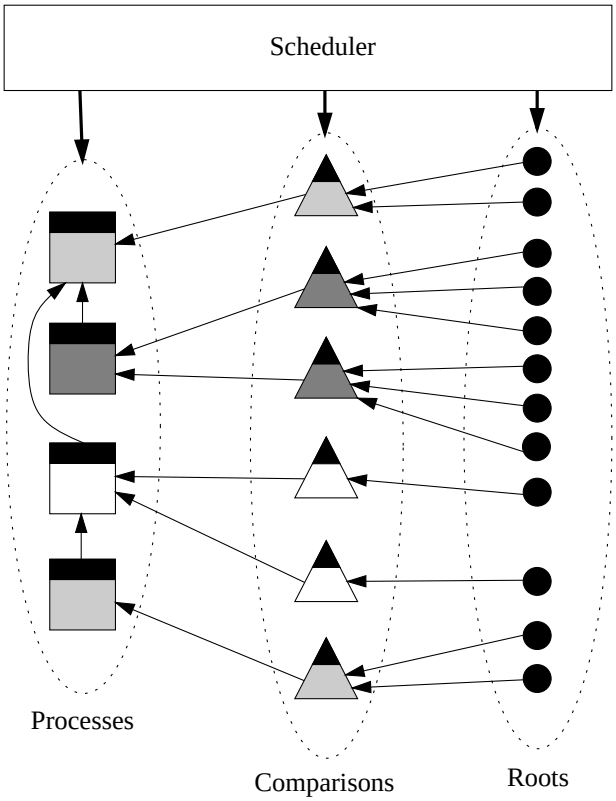

Figure 1: Overview of the main classes of the framework. A user may create several kind of processes, all derived from a common base class (depicted with a black rectangle). Similarly, there should be one or more comparison classes, derived from a common base class (depicted with a black triangle).

References

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

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

[3] R. van Oostrum and Remco C. Veltkamp. Parametric Search Made Practical. To appear in *Proc. 18th Annu. ACM Symp. on Computational Geometry*, 2002.