A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment

Bartłomiej Jacek Kubica and Adam Woźniak

Institute of Control and Computation Engineering, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland

Abstract. Previous investigations of the authors surveyed the possibility of applying interval methods to seek the Pareto-front of a multicriterial nonlinear problem and an efficient algorithm has been proposed. With the advent of multi-core computer architectures algorithms should be modified to fit the multi-threading model. This allows to increase the efficiency by running the computations in parallel on several cores. The paper presents a multi-threaded variant of the previously developed algorithm, parallelized using the *Pthreads* library. Numerical results for a hard test problem are presented.

1 Introduction

It is well known that interval methods can be used as a precise and robust tool to solve nonlinear problems of various types (see e.g. [1]), in particular multicriterial optimization problems (see [3] and references therein). A multicriterial optimization problem is a problem of the following form:

$$\begin{aligned} &\min_x q_k(x) \qquad k = 1, \ldots, N \ , \\ &\text{s.t.} \\ &g_j(x) \leq 0 \qquad j = 1, \ldots, m \ , \\ &x_i \in [\underline{x}_i, \overline{x}_i] \qquad i = 1, \ldots, n \ , \end{aligned} \tag{1}$$

where decision variable $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$. In the sequel we shall denote the set of points satisfying the above conditions as X (the set of feasible points). Precisely, we seek the Pareto-set or Pareto-front of the above problem, i.e. the set of all non-dominated points $x \in X$ and the image of this set.

This paper reminds a previously developed algorithm [3] and describes its parallelization using the *Pthreads* library (see e.g. [4], [5]).

Basics of interval computations and SIVIA (Set Inversion Via Interval Analysis) algorithm can be found e.g. in [1].

2 The algorithm to approximate the Pareto-front

In [3] an algorithm to seek the Pareto-front has been proposed. It subdivides the criteria space in a branch-and-bound manner and inverts each of the obtained

sets using a variant of the SIVIA procedure. Some additional tools (like the componentwise Newton operator) are applied to speedup the computations.

The algorithm is expressed by the following pseudocode.

```
compute_Pareto-front (q(·), x^(0), ε_y, ε_x)
// q(·) is the interval extension of the function q(·) = (q_1, ..., q_N)(·)
// L is the list of quadruples (y, L_in, L_bound, L_unchecked)
y^(0) = q(x^(0));
L = { (y^(0), {}, {}, {x^(0)}) };
while (there is a quadruple in L, for which wid y ≥ ε_y)
    take this quadruple (y, L_in, L_bound, L_unchecked) from L;
    bisect y to y^(1) and y^(2);
    for i = 1, 2
        apply SIVIA with accuracy ε_x to quadruple (y^(i), L_in, L_bound, L_unchecked);
        if (the resulting quadruple has a nonempty interior, i.e. L_in ≠ ∅)
            delete quadruples that are dominated by ȳ^(i);
        end if
        insert the quadruple to the end of L;
    end for
end while
end compute_Pareto-front
```

Please note that it is sufficient to *break* the SIVIA procedure after finding a so-called interior subbox. This leads to two variants of our algorithm, as described in [3]: "breaking SIVIA" and "non-breaking SIVIA".

3 A multi-threaded variant

Threads are a most commonly used tool to parallelize computations in a shared-memory environment. In opposite to "heavy" processes threads run in a common address space – they can share some of the variables and data structures (and obviously have private ones, too).

In our implementation the list L from the algorithm is shared and each thread has an instance of the main `while` loop.

Obviously, operations of fetching a quadruple from L, inserting a quadruple to L and deleting dominated quadruples have to be synchronized. A single mutex (mutual exclusion lock) associated with the list is proper here.

A bit more complicated issues are connected with checking if all boxes have already been investigated or not – each thread has to check not only if the list is empty, but also if other threads have finished computations or not. A *conditional variable* is used there in the way described below.

We define a table `finish_thread[]` of booleans – each thread has a corresponding element, but the array is shared by all threads. We use a mutex (as always with the conditional variable) to synchronize operations on the array. Initially each element of the array is set to zero (i.e. "do not finish").

When a thread realizes that the queue of quadruples is empty, it sets its flag to true and checks if other threads did. If so, it resumes all the threads, using `pthread_cond_broadcast()` (so that they could terminate) and finishes the work. Otherwise it suspends the execution, using `pthread_cond_wait()`.

On the other hand when a thread adds a new quadruple to the queue, this thread `pthread_cond_signal()`'s this fact to one of the waiting threads.

And when a thread wakes up, it checks all flags in `finish_thread[]` once more and either terminates or resets its own flag and continues work.

4 Numerical experiments

We shall present results for a test problem, used in [2]. It is a good benchmark for multicriterial optimization problems, because minimized functions are complicated and its Pareto-front and Pareto-set are both nonconnected.

$$\begin{aligned}\min_{x_1,x_2}\Big(q_1(x_1,x_2) &= -\big(3\cdot(1-x_1)^2\cdot\exp(-x_1^2-(x_2+1)^2)-10\cdot\big(\frac{x_1}{5}-x_1^3-x_2^5\big)\times\\ &\times\exp(-x_1^2-x_2^2)-3\exp(-(x_1+2)^2-x_2^2)+0.5\cdot(2x_1+x_2)\big),\\ q_2(x_1,x_2) &= -\big(3\cdot(1+x_2)^2\cdot\exp(-x_2^2-(1-x_1)^2)-10\cdot\big(-\frac{x_2}{5}+x_2^3+x_1^5\big)\times\\ &\times\exp(-x_2^2-x_1^2)-3\exp(-(2-x_2)^2-x_1^2)\big)\Big),\\ x_1,x_2 &\in[-3,3]\ .\end{aligned}\tag{2}$$

Table 1. Numerical results for test problem (2), non-breaking SIVIA variant of the algorithm, $\varepsilon_y = 0.2$, $\varepsilon_x = 0.001$

	Intel Core Duo		Intel Core 2 Quad		
number of threads $= N$	1	2	1	2	4
computational time $= T(N)$	47m3.21s	25m42.57s	15m40.688s	8m49.195s	4m42.419s
speedup $= T(1)/T(N)$	1	1.830198	1	1.777583	3.330824
criterion evals.	18591128	19237846	18591128	19108467	19713660
criterion grad evals.	15089688	15692644	15089688	15578848	16013148
bisections in criteria space	441	477	441	446	458
bisections in decision space	1689116	1755230	1689116	1737904	1786468
boxes deleted by monot. test	1	1	42881	43545	44420
boxes deleted by N_{cmp}	142765	144599	142765	144054	150041
resulting quadruples	174	178	174	177	190
resulting interior boxes	284989	291402	284989	289237	312948
resulting boundary boxes	424703	434670	424703	431585	469048
resulting unchecked boxes	0	0	0	0	0

Table 2. Numerical results for test problem (2), breaking SIVIA variant of the algorithm, $\varepsilon_y = 0.2$, $\varepsilon_x = 0.001$

	Intel Core Duo		Intel Core 2 Quad		
number of threads $= N$	1	2	1	2	4
computational time $= T(N)$	1m5.365s	34.278s	21.652s	11.790s	5.963s
speedup $= T(1)/T(N)$	1	1.906908	1	1.836472	3.631058
criterion evals.	561510	568108	561510	570600	567541
criterion grad evals.	292146	296104	292146	297474	297538
bisections in criteria space	440	448	440	450	452
bisections in decision space	60173	60917	60173	61242	60987
boxes deleted by monot. test	6460	6551	6460	6590	6634
boxes deleted by N_{cmp}	15648	15851	15648	15949	15722
resulting quadruples	173	176	173	176	179
resulting interior boxes	245	249	245	249	254
resulting boundary boxes	16300	16545	16300	16631	16782
resulting unchecked boxes	3676	3740	3676	3742	3807

Previous research confirmed that interval methods are well suited to approximate the Pareto-front of a multicriterial optimization problem. Now a parallel version of the algorithm was presented.

The speedup seems to be quite promising, especially for the "breaking SIVIA" variant of the algorithm, were first iterations when we do not have enough quadruples for all threads (and hence computations are not parallelized yet) are cheap. As we can see in the tables, for 4 cores (and 4 threads) the speedup was over 3.33 for the "non-breaking SIVIA" variant and over 3.63 for "breaking SIVIA" one, which is quite a good result.

It is worth noting that the parallelization seems to work a bit better for older Intel Core architecture than for Intel Core 2 – the difference is only about 7%, but it does not seem to be accidental.

The paper is going to describe also several important details of the implementation and present more computational results.

References

1. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. Springer, London, (2001).
2. Kim, I. Y., de Weck, O. L.: Adaptive weighted-sum method for bi-objective optimization: Pareto front generation. Structural and Multidisciplinary Optimization **29** (2005) 149–158.
3. Kubica, B.J., Woźniak, A.: Interval methods for computing the Pareto-front of a multicriterial problem. presented at PPAM 2007 Conference, Gdansk (2007).
4. Linux Tutorial: POSIX Threads, `http://www.youlinux.com/TUTORIALS/LinuxTutorialPosixThreads.html` .
5. POSIX Threads Programming, `https://computing.llnl.gov/tutorials/pthreads` .
6. C-XSC library, `http://www.xsc.de` .