Paper

Optimization of the Multi-Threaded Interval Algorithm for the Pareto-Set Computation

Bartłomiej J. Kubica and Adam Woźniak

Abstract—Previous investigations of the authors surveyed the possibility of applying interval methods to seek the Pareto-front of a multicriterial nonlinear problem. An efficient algorithm has been proposed and its implementation in a multi-core environment has been done and tested. This paper has two goals. First one is to tune the developed algorithm to increase the speedup of the multi-threaded variant. The second one is to extend the algorithm to compute not only the Pareto-front (in the criteria space), but also the Pareto-set (in the decision space). Numerical results for suitable test problems are presented.

Keywords—interval computations, multicriterial analysis, multi-threaded programming, Pareto set, POSIX threads, shared-memory parallelization.

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, e.g., [2]–[4]). A multicriterial optimization problem is commonly encountered in practical applications (e.g., [5]–[7]). It is a problem of the following form:

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

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 and Pareto-front of the above problem, i.e., the set of all non-dominated points $x \in X$ and the image of such set.

In this paper we recall a previously developed algorithm [4] and its parallelization using the *Pthreads* library [8]. Than, we try to optimize the parallel version to obtain high performance.

2. Basics of Interval Computations

Now, we shall define some basic notions of intervals and their arithmetic. We follow a widely acknowledged standards (cf., e.g., [1], [9], [10]).

We define the (closed) interval $[\underline{x}, \overline{x}]$ as a set $\{x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x}\}$. We denote all intervals by brackets; open ones will be denoted as $]\underline{x}, \overline{x}[$ and partially open as: $[\underline{x}, \overline{x}[$, $]\underline{x}, \overline{x}]$. (We prefer this notation to using the parenthesis that are used also to denote sequences, vectors, etc.)

Following [11], we use boldface lowercase letters to denote interval variables, e.g., $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbb{IR}$ denotes the set of all real intervals.

We design arithmetic operations on intervals so that the following condition was fulfilled: if we have $\odot \in \{+,-,\cdot,/\}$, $a \in \mathbf{a}$, $b \in \mathbf{b}$, then $a \odot b \in \mathbf{a} \odot \mathbf{b}$. The actual formulae for arithmetic operations (see, e.g., [1], [9], [10]) are as follows:

$$\begin{aligned} [\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \ , \\ [\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \ , \\ [\underline{a}, \overline{a}] \cdot [\underline{b}, \overline{b}] &= [\min(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}), \max(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab})] \ , \\ [\underline{a}, \overline{a}] \ / \ [\underline{b}, \overline{b}] &= [\underline{a}, \overline{a}] \cdot [1/\overline{b}, 1/\underline{b}] \ , \qquad 0 \notin [\underline{b}, \overline{b}] \ . \end{aligned}$$

Links between real and interval functions are set by the notion of an *inclusion function* (see, e.g., [1]); also called an *interval extension* (e.g., [10]).

Definition 1: A function $\mathsf{f}: \mathbb{IR} \to \mathbb{IR}$ is an *inclusion function* of $f: \mathbb{R} \to \mathbb{R}$, if for every interval $\mathbf{x}$ within the domain of f the following condition is satisfied:

$$\{f(x) \mid x \in \mathbf{x}\} \subseteq \mathsf{f}(\mathbf{x}) \ . \tag{2}$$

The definition is analogous for functions $f: \mathbb{R}^n \to \mathbb{R}^m$.

When computing interval operations, we can round the lower bound downward and the upper bound upward. This will result in an interval that will be a bit overestimated, but will be *guaranteed to contain the true result of the real-number operation*.

The quality of an interval approximation is often measured by the width of an interval, $\operatorname{wid}\mathbf{x} = \overline{x} - \underline{x}$.

3. The Algorithm to Approximate the Pareto-Set

In [4] an algorithm to seek the Pareto-front has been proposed. It subdivides the criteria space in a branch-and-bound (b&b) manner and inverts each of the obtained sets using a variant of the SIVIA procedure (i.e., set inversion via interval analysis, see [12]). Some additional tools

(like the componentwise Newton operator) are applied to speedup the computations.
The algorithm is expressed by the following pseudocode.

Algorithm 1: Pseudocode of the algorithm

```
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 an interior subbox. This leads to two variants of our algorithm, as described in [4]: "breaking SIVIA" and "non-breaking SIVIA".
Also, It should be noted that, while the "non-breaking SIVIA" variant computes both the Pareto-front and Pareto-set, the "breaking" one leaves several boxes (from the decision space) unchecked. To compute the Pareto-set we have to add a "finishing" procedure, described later in the paper.

4. 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 related to 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.
We define a table `finish_thread[]` of booleans – each thread has a corresponding element, but the array is shared by all threads. Obviously there is 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, its `pthread_cond_signal()`signals it 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.

5. Changes to the Algorithm

5.1. How to Increase the Efficiency of the Parallel Algorithm?

There are two major problems that decrease performance of the parallel algorithm:

- threads have to wait for each other when manipulating shared resources (in our case – the list of quadruples);
- the parallelism causes that quadruples that in the serial variant would be deleted in initial iterations are unnecessarily processed by other threads.

To minimize the influence of the first problem we have to make all operations that have to be synchronized as quick as possible. In our case the list of quadruples is implemented as a unidirectional linked list with a shortcut to the last element. Consequently, insertion of an element at the end is quick. On the other hand retrieving the box for which $\operatorname{wid}\mathbf{y} \geq \varepsilon_y$ requires a linear search of the list.
A simple improvement (similar to the one used in interval unicriterial global optimization algorithms; see, e.g., [10]) is to use two separate lists: the list L of boxes that are still processed and a new list S of small boxes that are not going to be bisected anymore.
Now the operation of obtaining the first box from L is as efficient as inserting a box to its end. The only costly operation that remains is the procedure of deleting dominated boxes, but as the list is now divided in two parts, this procedure improves, too.
Obviously, the list S must have its own mutex to synchronize operations on it ("insert" and "delete dominated").

Still the second of previously mentioned problems remains an important drawback of the method. Boxes that are currently processed by one of the threads are not removed by the "delete dominated" procedure and are going to be uselessly processed for several iterations.
To deal with this problem we add a third shared resource – a queue (implemented by a table of length roughly equal to the number of threads) of criteria vectors used lately to delete boxes that they dominate. New quadruples to be processed are compared with the values in the list – dominated ones are rejected.
Obviously, operations on this queue have to be synchronized, but – as these operations often require reading only – a readers-writer lock is more sufficient than a mutex there. As each thread reads and writes from it alternatingly (precisely: two reads than one write), no starvation is possible.

5.2. What to Do with the Lists of Unchecked Boxes?

The procedure to finish the computation for remaining quadruples is simple. The ordinary SIVIA procedure can be used on them; only with the "breaking SIVIA" flag unset.
What is more interesting is the parallelization of this part of the program. Two models were used to create threads for this computation:

- "many finishing threads" – the main thread iterates through the list L and creates a specific thread to finish the computations for each of the elements;
- "N finishing threads" – a given number of threads are created; they iterate through the list simultaneously and finish computations for different elements.

Obviously, both variants require proper extensions to the structure of elements, stored in L:

- in the first case we have to add a field to store tid of the finishing thread;
- in the second case we add to each element a flag finished and a mutex to protect it.

6. Numerical Experiments

Results for two test problems are going to be presented. The first one is constrained, but seems to be simple:

$$\begin{aligned}
&\min_{x_1,x_2}\Big(q_1(x_1,x_2) = -(5x_1+12x_2-x_1^2-x_2^2)\ ,\\
&q_2(x_1,x_2) = -(x_1+x_2)\Big)\ ,\\
&\text{s.t.}\\
&-2x_1-x_2+12\le 0\\
&-x_1+x_2-2\le 0\\
&4x_1-2x_2-47\le 0\\
&x_1,x_2\in[0,50]\ .
\end{aligned} \tag{3}$$

The second one is taken form [13]. It is a good benchmark for multicriterial optimization problems, because minimized functions are complicated and its Pareto-front and Pareto-set are both nonconnected (suitable figures are presented in [4]):

$$\begin{aligned}
\min_{x_1,x_2}\Big(q_1(x_1,x_2) &= -\big(3(1-x_1)^2\exp(-x_1^2-(x_2+1)^2)\\
&\quad -10\big(\frac{x_1}{5}-x_1^3-x_2^5\big)\exp(-x_1^2-x_2^2)\\
&\quad -3\exp(-(x_1+2)^2-x_2^2)\\
&\quad +0.5(2x_1+x_2)\big),\\
q_2(x_1,x_2) &= -\big(3(1+x_2)^2\exp(-x_2^2-(1-x_1)^2)\\
&\quad -10\big(-\frac{x_2}{5}+x_2^3+x_1^5\big)\exp(-x_2^2-x_1^2)\\
&\quad -3\exp(-(2-x_2)^2-x_1^2)\big)\Big),\\
x_1,x_2\in[-3,3]\ .&
\end{aligned} \tag{4}$$

Due to the nondeterministic nature of parallel computations, results for four runs are presented for each of the multi-threaded variants.

Table 1
Problem (3), $N=1$, $\varepsilon_y=10^{-3}$, $\varepsilon_x=10^{-4}$

Alg. variant	Non-breaking	Breaking
old $T(1)$	182	23
new $T(1)$	181	24
f	18319573	3274377
∇f	15517892	921836
g	405435	209820
∇g	559212	414915
bis y	159127	47625
bis x	279605	155730
L	91309	27434
L_{in}	1	486
L_{bound}	433627	202645

Table 2
Problem (3), $N=2$, old, breaking SIVIA, $\varepsilon_y=10^{-3}$, $\varepsilon_x=10^{-4}$

No.	1	2	3	4
$T(2)$	44	48	56	33
f	3274377	3313486	3274502	3274397
∇f	921838	932688	921960	921842
g	209820	211848	209833	209820
∇g	414915	419485	414972	414915
bis y	47625	48588	47627	47626
bis x	155730	157169	155739	155730
L	27440	27986	27443	27443
L_{in}	487	488	488	489
L_{bound}	202689	205345	202718	202714

The program was implemented in C++, using C-XSC 2.2.1 library [14] for interval computations. The parallelization was done using POSIX threads [15].
Computations were performed on a machine with Intel S775 Core 2 Quad Q6600 2.4 GHz processor and 2 GB RAM, under control of Linux Slackware 12.0 operating system (with the 2.6.21.5-smp kernel). The GCC compiler was used in version 4.1.2.

Table 3
Problem (3), $N = 4$, old, breaking SIVIA, $\varepsilon_y = 10^{-3}$, $\varepsilon_x = 10^{-4}$

No.	1	2	3	4
$T(4)$	46	68	86	96
f	3706197	3706826	3274545	3275720
∇f	1037810	1038116	921864	922546
g	231104	231214	209820	209960
∇g	463436	463596	414915	415246
bis y	58882	58880	47629	47636
bis x	170578	170665	155730	155836
L	33728	33722	27447	27444
L_{in}	491	488	494	487
L_{bound}	231764	231732	202752	202716

Table 4
Problem (3), $N = 2$, new, breaking SIVIA, $\varepsilon_y = 10^{-3}$, $\varepsilon_x = 10^{-4}$

No.	1	2	3	4
$T(2)$	20	21	16	18
$T(1)/T(2)$	1.20	1.14	1.5	1.33
f	3644854	3624613	3246557	3222754
∇f	1022666	1021852	917838	912122
g	229376	229735	209855	209144
∇g	456835	456669	412969	410852
bis y	58306	58026	47919	47119
bis x	169214	169555	155630	155270
L	32958	32741	27173	26777
L_{in}	483	489	477	486
L_{bound}	225999	223533	199139	197038

Tables 1–11 present a few variants of the algorithm:

- a single-threaded program, using breaking or non-breaking SIVIA algorithm variants;
- "old" multi-threaded implementations of breaking or non-breaking SIVIA algorithm variants; they do not use modifications presented in this paper;
- "new" multi-threaded implementations of breaking or non-breaking SIVIA algorithm variants; they use the modifications presented in this paper.

The "new" implementations have "finishing threads" as described above; their number may be equal to the number of threads that execute the b&b method (2 or 4) or there might be an "indefinite number of finishing threads", which is explicitly marked then.

Table 5
Problem (3), $N = 4$, new, breaking SIVIA, $\varepsilon_y = 10^{-3}$, $\varepsilon_x = 10^{-4}$

No.	1	2	3	4
$T(4)$	12	14	12	15
$T(1)/T(4)$	2.00	1.71	2.00	1.60
f	3222390	3637486	3247590	3616950
∇f	912900	1022034	919978	1022494
g	209513	229731	210574	230025
∇g	411207	456761	414036	457160
bis y	47053	58121	47830	57734
bis x	155442	169501	156188	169853
L	26544	32640	26943	32428
L_{in}	490	475	488	484
L_{bound}	195903	224062	197702	222481

Table 6
Problem (3), $N = 4$, new, non-breaking SIVIA, $\varepsilon_y = 10^{-3}$, $\varepsilon_x = 10^{-4}$

No.	1	2	3	4
$T(4)$	273	200	213	202
f	18539981	18445743	18408647	18451755
∇f	15924704	15814380	15763692	15848002
g	405435	405435	405435	405435
∇g	559212	559212	559212	559212
bis y	157672	157538	157668	157481
bis x	279605	279605	279605	279605
l	89164	89018	89102	88866
L_{in}	1	1	1	1
L_{bound}	420030	420318	420451	417555

Please note that differences between the "old" and "new" variant of the algorithm rely on synchronization primitives and data structures management only, so they do not affect the number of criterion functions evaluations, gradients evaluations, etc. Only times of computation differ as it can be seen in Tables 1 and 7.
Notation for Tables 1–11 is as follows:

- $T(N)$ – computation time in seconds (for N threads),
- f – number of criterion evaluations,
- ∇f – number of criterion gradient evaluations,
- g – number of constraints evaluations,
- ∇g – number of constraints gradients evaluations,
- bis y – number of bisections in the criteria space,
- bis x – number of bisections in the decision space,
- L – number of resulting quadruples,
- L_{in} – number of resulting interior boxes,
- L_{bound} – number of resulting boundary boxes.

In captions of the tables we write:

- test problem number;
- "old" – for the algorithm described in [8] or "new" – for the modified method, presented here;
- "$N = \ldots$" for the number of threads, adding "indefinite" if the number of finishing threads was such;
- "breaking SIVIA" or "non-breaking SIVIA";
- accuracies: ε_y and ε_x.

For Tables 2, 3 and 6 a slowdown was obtained instead of a speedup, so we do not compute any speedup $T(1)/T(N)$ for them.

Table 7
Problem (4), $N = 1$, old, $\varepsilon_y = 0.2$, $\varepsilon_x = 10^{-3}$

Alg. variant	Non-breaking	Breaking
old $T(1)$	941	254
new $T(1)$	942	251
f	18591128	5786259
∇f	15089688	3358486
bis y	441	440
bis x	1689116	819561
L	174	173
L_{in}	284989	301741
L_{bound}	424703	398416

Table 8
Problem (4), $N = 4$, indefinite, new, breaking SIVIA, $\varepsilon_y = 0.2$, $\varepsilon_x = 10^{-3}$

No.	1	2	3	4
$T(4)$	129	125	126	146
$T(1)/T(4)$	1.95	2.01	1.99	1.92
f	6070586	5914523	5912838	6834256
∇f	3527204	3434490	3431324	3957086
bis y	451	462	448	480
bis x	860940	837719	837227	966762
L	181	178	178	203
L_{in}	315906	308057	307955	352962
L_{bound}	418477	406901	407869	476708

Table 9
Problem (4), $N = 2$, new, breaking SIVIA, $\varepsilon_y = 0.2$, $\varepsilon_x = 10^{-3}$

No.	1	2	3	4
$T(2)$	130	129	130	131
$T(1)/T(2)$	1.93	1.95	1.93	1.92
f	5896384	5786532	5834466	5929859
∇f	3422286	3359012	3386998	3441636
bis y	441	442	441	441
bis x	835183	819592	826520	839983
L	177	173	175	178
L_{in}	307555	301741	304292	309399
L_{bound}	406781	398416	402308	409390

Table 10
Problem (4), $N = 4$, new, non-breaking SIVIA, $\varepsilon_y = 0.2$, $\varepsilon_x = 10^{-3}$

No.	1	2	3	4
$T(4)$	299	293	275	291
$T(1)/T(4)$	3.15	3.21	3.42	3.23
f	20239650	20132048	18648308	19925301
∇f	16244024	16147668	15089772	15997082
bis y	488	494	451	484
bis x	1819498	1807109	1701210	1793874
L	200	201	182	200
L_{in}	337125	339912	298077	334196
L_{bound}	500867	507761	446478	498749

Table 11
Problem (4), $N = 4$, new, breaking SIVIA, $\varepsilon_y = 0.2$, $\varepsilon_x = 10^{-3}$

No.	1	2	3	4
$T(4)$	68	67	67	66
$T(1)/T(4)$	3.69	3.75	3.75	3.80
f	6169467	5967139	6055588	5851089
∇f	3579196	3463062	3510842	3396582
bis y	454	441	452	445
bis x	874123	845219	856911	828741
L	185	180	183	175
L_{in}	319963	311211	314298	304949
L_{bound}	427422	411917	418824	403354

7. Results Analysis

As it was stated in [4], our algorithm can compute the approximation of the whole Pareto-front in nonconnected case, compared to pointwise approximation with only 15 points, obtained by classical methods (see [13], [16]). It means that potentialities of the proposed algorithm are interesting and we hope that it can be used to solve practical problems.

As we can see in Tables 1–3, the old variant of the parallel algorithm, presented in [4] achieved no speedup for test problem (3). It is surprising, but apparently, the penalty for synchronization is too large. Fortunately, due to changes made to the program a speedup is obtained (Tables 4 and 5).

The "indefinite number of finishing threads" variant seems inefficient – on 4 processors its performance was comparable to "N finishing threads" on 2 processors for problem (4). Although the operating system managed to schedule the large number of threads properly, it clearly consumed too much resources. Clearly, as thread creation and joining is relatively expensive on today architectures, it seems optimal to have the number of threads (approximately) equal to the number of processors/cores.

For problem (3) it did not work at all – it required too much memory for a single process. Please note, also, that as this variant was easy to implement in *Pthreads*, it would be very difficult to implement, e.g., in classical *OpenMP* (older than version 3.0) that does not use tasks.

Changes made to previously created algorithm resulted in speedup of the parallel implementation. While the older version achieved no speedup for problem (3), the modified did. Anyway, the speedup was not as great as for problem (4).
It is also worth noting that the "breaking SIVIA" variant of the algorithm occurred to parallelize better than traditional, "non-breaking SIVIA" one. If SIVIA is broken after finding an interior box, much work is moved from the branch-and-bound method (which requires relatively much synchronization between concurrent threads) to the "finishing" part of the algorithm which requires no synchronization and can even be classified as "embarrassingly parallel".

8. Conclusions

Interval methods seem to be well suited to approximate the Pareto-set of a multicriterial optimization problem. Efficient parallelization, based on POSIX threads, targeted for a mutli-core environment has been proposed by authors.
Thanks to proper use of several synchronization primitives and suitable algorithm tuning, the program parallelizes well on 4 cores, allowing speedup over 3.82 for test problem (4). Testing the algorithm on higher number of cores will be subject to our future research.

Acknowledgment

The research has been supported by the Polish Ministry of Science and Higher Education under grant N N514 416934.

References

[1] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*. London: Springer, 2001.
[2] V. Barichard and J. K. Hao, "Population and Interval Constraint Propagation Algorithm", in *Second International Conference on Evolutionary Multi-Criterion Optimization (EMO 2003), Faro, Portugal, April 8–11, 2003*, LNCS, vol. 2632. Berlin-Heidelberg: Springer, 2003, pp. 81–101.
[3] G. R. Ruetsch, "An interval algorithm for multi-objective optimization", *Struct. Multidiscip. Opt.*, vol. 30, no. 1, pp. 27–37, 2005.
[4] B. J. Kubica and A. Woźniak, "Interval methods for computing the Pareto-front of a multicriterial problem", in *The Seventh International Conference on Parallel Processing and Applied Mathematics PPAM 2007, Gdańsk, Poland, September 2007*, LNCS, vol. 4967. Berlin-Heidelberg: Springer, 2008, pp. 1382–1391.
[5] M. Marks and E. Niewiadomska-Szynkiewicz, "Multiobjective approach to localization in wireless sensor networks", *J. Telecommun. Inform. Technol.*, no. 3, pp. 59–67, 2009.
[6] W. Ogryczak, "Reference point method with importance weighted partial achievements", *J. Telecommun. Inform. Technol.*, no. 4, pp. 17–25, 2008.
[7] C. Gomes da Silva and J. C. N. Clímaco, "A note on the computation of ordered supported non-dominated solutions in the bi-criteria minimum spanning tree problems", *J. Telecommun. Inform. Technol.*, no. 4, pp. 11–15, 2007.
[8] B. J. Kubica and A. Woźniak, "A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment", in *PARA 2008 Conf.*, Trondheim, Norway, 2008.
[9] E. Hansen, *Global Optimization Using Interval Analysis*. New York: Marcel Dekker, 1992.
[10] R. B. Kearfott, *Rigorous Global Search: Continuous Problems*. Dordrecht: Kluwer, 1996.
[11] R. B. Kearfott, M. T. Nakao, A. Neumaier, S. M. Rump, S. P. Shary, and P. van Hentenryck, "Standardized notation in interval analysis", http://www.mat.univie.ac.at/ neum/software/int/notation.ps.gz
[12] L. Jaulin and E. Walter, "Set inversion via interval analysis for nonlinear bounded-error estimation", *Automatica*, vol. 29, no. 4, pp. 1053–1064, 1993.
[13] I. Y. Kim and O. L. de Weck, "Adaptive weighted-sum method for bi-objective optimization: Pareto front generation", *Struct. Multidiscip. Opt.*, vol. 29, no. 2, pp. 149–158, 2005.
[14] "C-XSC library", http://www.xsc.de
[15] "POSIX threads programming", https://computing.llnl.gov/tutorials/pthreads
[16] E. Zitzler, M. Laumanns, and M. Thiele, "SPEA2: improving the strength Pareto evolutionary algorithm for multiobjective optimization", in *Evolutionary Methods for Design Optimization and Control*, K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. Fogarty, Eds. Barcelona: CIMNE, 2002.

Bartłomiej Jacek Kubica received his Ph.D. in computer science in 2006 from the Warsaw University of Technology (WUT), Poland. Since 2005 he is employed at WUT. Currently he works as an Assistant Professor in Complex Systems Group. He coorganizes interval sessions at PPAM conferences and organizes at PARA. He co-authored a book on parallel programming and wrote several papers and presentations. His research interests focus on interval methods, parallel computations and optimization algorithms.
e-mail: bkubica@elka.pw.edu.pl
Institute of Control and Computation Engineering
Warsaw University of Technology
Nowowiejska st 15/19
00-665 Warsaw, Poland

Adam Woźniak received Ph.D. in control science in 1975 from the Warsaw University of Technology (WUT), Poland. He is employed at the Institute of Control and Computation Engineering of WUT since 1970. Now he is a reader in Systems Control Division, Complex Systems Group. His research interests include control of complex systems, robot control, decision support systems, multicriteria optimization, game theory, multiagent systems including mechanism design and auctions, interval methods applications.
e-mail: A.Wozniak@ia.pw.edu.pl
Institute of Control and Computation Engineering
Warsaw University of Technology
Nowowiejska st 15/19
00-665 Warsaw, Poland