Interval Computations
No 3, 1994

Parallel Algorithms for Interval Computations: An Introduction

Vladik Kreinovich and Andrew Bernat

The survey of parallel algorithms and parallelization methods used in interval computations including reasons and profits of using parallel computations, when solving tasks in interval way.

Параллельные алгоритмы для интервальных вычислений. Введение

В. Крейнович, Э. Бернат

Обзор параллельных алгоритмов и методов параллелизации, используемых в интервальных вычислениях, с объяснением причин и преимуществ использования параллельных вычислений при решении задач интервальными методами.

Conte

We all love parallel computations
B'cause for them, there are no limitations!
And to use world-wide net
From an every man's flat
That's indeed the Great Dream of all nations.

Contents

1 Par of glin

Let's face it essary, but)

There is problems, important t

Necessa ficult. Exci interval me that gives a trick there

The ma ers that pa our enthusi

In this We have tl reports in a a survey. to make it missed or a

Let us l

[1]We made ever made us we present hin

1 Parallelism: one more (ya-a-awn) aspect of interval computation or an exciting glimpse into the future?

Let's face it, there are many things in interval computation that are (necessary, but) dull.

There is a universe of numerical problems out there. For many of these problems, existing algorithms do not give a guaranteed estimate. It is important to find an interval version whose results will be guaranteed.

Necessary? Oh, yes. Technically difficult? Often extraordinarily difficult. Exciting? But, honestly, it is difficult to get excited about a new interval method of solving, say, stochastic integro-differential equations that gives accuracy $n^{-0.6}$ instead of $n^{-0.5}$ (unless of course, there is a new trick there or an exciting application)[1].

The main purpose of this Editors' introduction is to persuade the readers that parallel computations *are* exciting. We want the reader to share our enthusiasm.

In this Introduction, we have tried to be as up-to-date as possible. We have therefore included conference abstracts and available technical reports in addition to published papers. However, this introduction is not a survey. To be more accurate, it is not yet a survey. We would love to make it more comprehensive, and we welcome any references that we missed or any suggestions for enhancing its usefulness.

Let us begin!

[1] We made this example up, carefully avoiding any mentioning of any real problem that has ever made us yawn. If by accident there is someone who has actually proved such a theorem, we present him/her with our most sincere (yawn, sorry) apologies.

2 Why are interval computations necessary?

2.1 A (brief) general overview of the problems that our world faces

Before we start convincing the reader that parallel methods are necessary for interval computations, let's first describe why interval computations are necessary in the first place.

Again, this is not a survey, so we do not enumerate all possible reasons, but we will try not to miss the main ones.

Crudely speaking, all problems that we are solving in real life can be divided into two big groups: *analysis* and *synthesis*.

- *Analysis* means that we already have an object, and we are interested in some properties of this object.
- *Synthesis* means that we are not satisfied with the existing objects, and we would like to design new objects with desired properties.

Both types of problems use computations.

2.2 Computational aspects of analysis, and where interval computation comes into the picture

Analysis means that we are interested in some properties of a known object.

Some of these properties, we can simply measure (like a body temperature). No computation is necessary here. However, such measurements are never absolutely accurate. Therefore, the result $\tilde{x}$ of measuring x can differ from the actual value of x. How large can an error $\Delta x = \tilde{x} - x$ be? Manufacturers of measuring instruments guarantee some *accuracy* Δ; this means that the error will never exceed Δ: $|\Delta x| \leq \Delta$. So, if our measurement result is $\tilde{x}$, then the possible values of $x = \tilde{x} - \Delta x$ form an *interval* $[\tilde{x} - \Delta, \tilde{x} + \Delta]$.

Sometimes, in addition to the accuracy, the manufacturer guarantees some statistical characteristics of the measurement error; typically a standard deviation. This situation is well-developed in measurement theory (see, e.g., [86]).

Some quantities we can simply measure. But for many others quantities, it is either impossible or too costly to measure them directly. Such situations are very frequent. For example, practically all the numbers that excite the readers of a popular science journal cannot be measured directly. There is no ruler to measure the distance between us and a quasar, no scales to weigh the Earth or our Galaxy, no speedometer to measure directly the speed of elementary particles.

Since we cannot simply measure, we need to compute such values. All these values are measured *indirectly*: we measure several other quantities $x_1, \ldots, x_n$ that are related to the desired one y, and then we reconstruct the value of y from the results $\tilde{x}_i$ of measuring x_i. In other words, we have an algorithm f that takes the values $\tilde{x}_i$ and returns an estimate $\tilde{y} = f(\tilde{x}_1, \ldots, \tilde{x}_n)$. This estimate is called a result of indirect measurement. And here comes the problem:

The problem is to estimate the error of this estimate.

For example, in case we know the accuracies Δ_i with which we measured x_i (i.e., if we know the intervals $[\tilde{x}_i - \Delta_i, \tilde{x}_i + \Delta_i]$ of possible values of x_i), then we would like to know the interval of possible values of y. This is the basic problem of interval computation with which the entire field started (see, e.g., [64]):

We know:

1) n intervals X_i;

2) an algorithm f that transforms n real numbers $x_1, \ldots, x_n$ into a real number $y = f(x_1, \ldots, x_n)$.

We are interested in estimating the interval

$$f(X_1, \ldots, X_n) = \{y \mid y = f(x_1, \ldots, x_n) \text{ for some } x_1 \in X_1, \ldots, x_n \in X_n\}.$$

The first algorithms of interval computation were designed to solve this problem, i.e., either to find $f(X_1, \ldots, X_n)$, or, if this is not exactly possible, at least find an interval F that contains $f(X_1, \ldots, X_n)$.

Sometimes, the relationship between y and x_i is given only implicitly. We just considered the case when we already have an algorithm that transforms x_i into y. In some real-life situations, we do not have such an algorithm. Instead, we know equations that relate y and x_i. So, to determine y, we must solve this system of equations.

So, suppose that we know intervals X_i of possible values of x_i, and an equation that relates x_i and y, and we want to find an interval Y of possible values of y. In such a situation, one possibility is first to find an algorithm that solves the equation, i.e., that transforms x_i into y, and then apply some known method of estimating F to this algorithm.

But in many cases (even for linear equations), it turns out that one can compute the interval of possible values of y directly from the system of equations, and end up with a better estimate and/or with a faster algorithm.

In some cases, the situation is even more complicated. In our description of the problem, we considered the case when we know the exact relationship between x_i and y. In other words, we considered cases in which either $y = f(x_1, \ldots, x_n)$ for some known f, or $F(x_1, \ldots, x_n, y) = 0$ for some known F.

But in many cases, we know the relationship between x_i and y only approximately (i.e., $y \approx f(x_1, \ldots, x_n)$, but generally speaking, $y \neq f(x_1, \ldots, x_n)$). In such cases, even if we knew the exact values of x_i, we would still get an estimate $\tilde{y} = f(x_1, \ldots, x_n)$ that is different from the desired value y.

This situation occurs, e.g., when we are interested in a physical quantity z that is not uniquely determined by x_i (in the sense that it is possible to have two situations with exactly the same values of $x_1, \ldots, x_n$ but different values of z). In such situations, we cannot estimate z from x_i, but what we can estimate is the probabilities for different value of z or other statistical characteristics y of Y. Such a characteristic y is already uniquely determined by x_i.

In some cases, there exist algorithms that estimate y exactly. But often, no ready-made algorithm is known that will compute the value y of the desired characteristic for a given $x_1, \ldots, x_n$. In such cases, we can use the following general method (called the Monte-Carlo approach):

1) we use a computer to simulate the stochastic dependency between x_i and z;

2) we run this simulation program several times; as a result, we obtain a finite set of values $z^1, \ldots, z^N$; from the statistical viewpoint, this set is a sample from the population distributed according to the desired probability distribution;

3) we then use standard methods of mathematical statistics to estimate the desired parameter y of this distribution from the sample $\{z^1, \ldots, z^N\}$.

As a result, we obtain an estimate $f(x_1, \ldots, x_n)$ for y. This f is a statistical estimate that is based on using a finite sample. Therefore, f is only an approximate estimate.

Monte-Carlo algorithms are used not only for statistical problems, but for deterministic problems as well. Monte-Carlo method sounds simple and fast. It actually is (in many cases) simple and fast. That is why it is used not only when we do not have an algorithm for computing $y = f(x_1, \ldots, x_n)$, but also when such an algorithm exists, but is too complicated. For example, when we know that y is a multi-dimensional integral $I = \int_\Omega F(\vec{y})\, d\vec{y}$ over some area Ω, then instead of approximating the interval by a multi-dimensional integral sum, we simulate a random vector that is uniformly distributed on Ω. For this random variable, the integral is equal to the volume of this area $V(\Omega)$ multiplied by a mathematical expectation M of $F(\vec{y})$. The mathematical expectation is known to be equal to the limit of the arithmetic average $(1/N)\big(F(\vec{y}^1) + \cdots + F(\vec{y}^N)\big)$ when $N \to \infty$. Therefore, to estimate M, we run the simulation program several times, compute $F(\vec{y}^k)$ for the resulting $\vec{y}^k$, and take the arithmetic average as the desired estimate for M.

Another application of a Monte-Carlo method to error estimation is even closer home: [46–48, 50] use the technique to solve the basic problem of interval computation.

2.3 Computational problems related to synthesis

Synthesis means that we want to design an object with desired properties. In computational terms, it means that we want to compute the design parameters $\vec{x}$ of the desired object. There are two main types of synthesis problems.

1. Problems in which we want to find the values $\vec{x}$ that satisfy some given properties. These properties are usually formulated in terms of equalities and inequalities, so, in mathematical terms, we are interested in solving an equation, or a system of equations and inequalities.

   For example, we want to find the parameters of the controller that make the plant stable.

   In this formulation, if there are several different sets of values that satisfy the given property, then any of these sets will do.

2. Problems in which we are looking for a vector $\vec{x}$ that not only satisfies the given constraints, but which is *optimal* in the sense that for $\vec{x}$, a given objective function $J(\vec{x})$ attains the biggest possible value.

In both cases, we must take into consideration the fact that the parameters are often determined by measurement and thus not exactly known. Therefore, methods have been developed that take interval uncertainty into consideration while solving equations or optimization problems. These methods are also an important part of interval computation.

3 Why parallelism is necessary for interval computation?

3.1 Interval computation is NP-hard

In the previous section, we formulated the basic problem of interval computation. Alas, life is tough. Even for the simplest functions f, this problem cannot be solved. To be more precise, [19] proves that this problem is infeasible (or, to use the precise mathematical term, NP-hard; see, e.g., [26]) even for polynomial f.

Another example of an intractable problem (in the general case) is solving a system of interval linear equations $\sum_j A_{ij} x_j = b_i$ ([49]) when we know only the intervals of possible values of A_{ij} and b_i, and we are interested in finding the intervals of possible values of x_i.

3.2 NP-hardness

That a problem U is NP-hard means that if there exists an algorithm that solves U in polynomial time, i.e., whose running time does not exceed some polynomial of the input length, then a polynomial-time algorithm would exist for practically all discrete problems such as propositional satisfiability problem, discrete optimization problems, etc. It is a common belief that for at least some of these discrete problems no polynomial-time algorithm is possible (this belief is formally described as $\mathrm{P} \neq \mathrm{NP}$). Thus there cannot be a polynomial time algorithm for the interval computation case (or else there would be for all discrete problems, which we don't believe). So, the fact that the problem is NP-hard means that no matter what algorithm we use, there will always be some cases for which the running time grows faster than any polynomial. Therefore, for these cases the problem is intractable.

In other words: if a problem is NP-hard, this means that no practical algorithm is possible that will solve all particular cases of this problem.

For interval computation this means that if we have an algorithm that inputs f and X_i and returns an interval $F \supseteq f(X_1, \ldots, X_n)$, then:

1) either this algorithm takes too long to compute F (sometimes, a time that grows exponentially with n); or

2) this algorithm overestimates, i.e., returns an interval F that is much larger than the minimal interval $f(X_1, \ldots, X_n)$.

In either case parallelism is necessary for interval computations:

1. If an algorithm takes too much time, then it is natural to run several computational steps simultaneously on several processors. Thus, the total running time will decrease.

2. Suppose now that an algorithm U (that is currently the best for a given problem) overestimates. This algorithm U is the best in

the sense that, e.g., among all known algorithms that compute F in reasonable time, this one produces the intervals F that are (in the majority of cases) the closest to the ideal interval $f(X_1, \ldots, X_n)$. Although F is the best in the majority of cases, in other cases, other algorithms can give better estimates F. So, it is natural to combine this "best" algorithm with one or several other algorithms, and produce the intersection of all the resulting estimates F as the estimate for $f(X_1, \ldots, X_n)$.

- If U is fast and there is time left, we can use this extra time to run some other algorithms and thus improve F.
- But when we have thus exhausted a given running time and still want to improve F, we have no other possibility but to run other algorithms in parallel.

Well, life is tough, but not that tough. We have argued that parallelism is *necessary* for interval computations. Let us now show that many problems of interval computations are easily parallelizable, so parallelism is *possible*. Before we do that, we issue one warning.

4 Warning: almost all algorithms are (theoretically) parallelizable, but this parallelization is not always practically usable

Parallelization is theoretically possible for virtually any algorithm. We argue as follows. Each algorithm is a sequence of computational steps. The final step consists of applying some elementary operation op to some previous results a and b: $f := op(a, b)$. If neither a nor b is given, but each is obtained by some series of computations, then this algorithm can be parallelized as follows: let one processor compute a, and at the same time let another one compute b; then, compute $f = op(a, b)$.

However, if one of the values a, b is given, then this method will not work. There are two possible cases here:

1. If both values are part of the input, i.e., they are both given, then this algorithm consists of a single elementary operation. So, its running time is equal to the time of one elementary operation. This algorithm is thus already as fast as possible, and of course, we cannot gain by parallelizing it.

2. Assume now that only one of the values is given. Then, the other value must be computed. We can apply the above idea to parallelize the computation of this second value and thus decrease the total running time.

The only case when this idea will not work is when we have an algorithm that at each stage, combines some computed result with a given value. There are a few real-life algorithms of this type:

- computing the sum $s = x_1 + \cdots + x_n$ of n given values;
- computing the product $s = x_1 \times \cdots \times x_n$ of n given values;
- computing the scalar (dot) product $s = x_1 \times y_1 + \cdots + x_n \times y_n$ of two given vectors $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_n)$.

All of these algorithms are also parallelizable (see, e.g., [36]). For example, we compute a sum using two processors:

1) divide the sum into two parts: from 1 to $n/2$, and from $n/2+1$ to n;
2) compute the first sum $y_1 = x_1 + \cdots + x_{n/2}$ on the first processor, and simultaneously compute the second sum $y_2 = x_{n/2+1} + \cdots + x_n$ on the second processor;
3) compute the final sum $s = y_1 + y_2$.

The problem with the above-described "theoretical parallelization" is that communication also takes some time; usually, the time for one communication step is much larger than the time for one computation step. Therefore, if by parallelizing, we save the time equivalent to one computation step, at the expense of one additional communication step, then overall, the running time will increase rather than decrease as we expected. Hence, such a parallelization makes no sense.

For example, if $f = (x_1 + x_2) \cdot (x_3 + x_4)$ (3 elementary operations), then according to the above-described idea, we can compute $a = x_1 + x_2$ and $b = x_3 + x_4$ in parallel (taking the time of one computational step), and then multiply these results (one more computational step). Totally, we need the time of two computational steps for computations (thus saving one). However, we would need one communication step, since in order to multiply the values a and b (that have been computed by different processors), we must bring them to one processor.

In view of the above warning, in the following text we will examine only reasonable parallelizations, those that actually save time.

5 Using parallelism in interval computation

5.1 f is linear: The simplest possible case

In this case, $y = c_1 x_1 + \cdots + c_n x_n$. If we know the intervals $X_i = [x_i^-, x_i^+]$ of possible values of x_i, then the interval Y of possible values of y is equal to $c_1 X_1 + \cdots + c_n X_n$, where cX and $X + Y$ are standard interval operations:

$$\begin{aligned} c[x^-, x^+] &= [cx^-, cx^+], \text{ if } c \geq 0 \\ c[x^-, x^+] &= [cx^+, cx^-], \text{ if } c < 0 \\ [a^-, a^+] + [b^-, b^+] &= [a^- + b^-, a^+ + b^+]. \end{aligned}$$

This problem is easily parallelizable (as in the similar non-interval problem, see above): if we have two processors available, then we do the following:

1) divide the sum into two parts: from 1 to $n/2$, and from $n/2+1$ to n;
2) compute the first sum $Y_1 = c_1 X_1 + \cdots + c_{n/1} X_{n/2}$ on the first processor, and simultaneously compute the second sum $Y_2 = c_{n/2+1} X_{n/2+1} + \cdots + c_n X_n$ on the second processor;
3) compute the final sum $Y = Y_1 + Y_2$.

As a result, we need only about half the time to compute the sum. If we have more processors, then we can apply the same trick to speed up the computation of each half-sum, etc.

Similar p
linear operat
and to multi

The abov
tations can a
multi-linear s
that mainly c

As an exa
of linear equ
course, use o
equations, pa

5.2 f is

Very frequent
be safely negl
is $\approx 1\%$ (i.e.,
this order are
can be safely
problem as fo

In our ma
accuracy Δ_i,
values of x_i. S
$\tilde{x}_i$ as $\tilde{x}_i = (1/$
element x_i of
$\tilde{x}_i - x_i$ is such

In these te
where $\Delta y = \tilde{y}$
$y = f(x_1, \ldots$
$\Delta y = f(\tilde{x}_1, \ldots$
assumed to b
series, and ne
Δx_i. As a re
$l = f_{,1} \Delta x_1 + \ldots$
$\frac{\partial f}{\partial x_i}$ at the poi

Similar parallelization techniques can be applied to computing multi-linear operations such as the scalar (dot) product of two interval vectors, and to multiplying two interval matrices (see, e.g., [6, 13, 37, 38, 51, 89]).

The above-described parallelization of linear and multi-linear computations can also speed up an arbitrary algorithm that contains linear or multi-linear steps. In particular, this speedup is dramatic for algorithms that mainly consist of such steps.

As an example of such algorithms, we can take the solution of *systems of linear equations* (see, e.g., [4, 61, 100, 111]; these parallelizations, of course, use other ideas as well). For the special class of such 3–diagonal equations, parallel algorithms were presented in [17, 18, 39, 40].

5.2 f is almost linear

Very frequently, the measurement errors are so small that their squares can be safely neglected (see, e.g., [86]). For example, if the measurement error is $\approx 1\%$ (i.e., ≈ 0.01), then the square of this error is ≈ 0.0001. Terms of this order are much smaller than 1%, and for all practical purposes, they can be safely neglected. Thus f is nearly linear and we may solve this problem as follows.

In our main example (indirect measurements), after measuring x_i with accuracy Δ_i, we get an interval $[x_i^-, x_i^+] = [\tilde{x}_i - \Delta_i, \tilde{x}_i + \Delta x_i]$ of possible values of x_i. So, if we are given an interval $[x_i^-, x_i^+]$, then we can reconstruct $\tilde{x}_i$ as $\tilde{x}_i = (1/2)(x_i^- + x_i^+)$, and Δ_i as $\Delta_i = (1/2)(x_i^+ - x_i^-)$. An arbitrary element x_i of this interval can be represented as $\tilde{x}_i - \Delta x_i$, where $\Delta x_i = \tilde{x}_i - x_i$ is such that $|\Delta x_i| \leq \Delta_i$.

In these terms, we are interested in finding the upper bound Δ for $|\Delta y|$, where $\Delta y = \tilde{y} - y$ (error of indirect measurement), $\tilde{y} = f(\tilde{x}_1, \ldots, \tilde{x}_n)$, and $y = f(x_1, \ldots, x_n)$. By the definition of Δx_i, $x_i = \tilde{x}_i - \Delta x_i$, therefore, $\Delta y = f(\tilde{x}_1, \ldots, \tilde{x}_n) - f(\tilde{x}_1 - \Delta x_1, \ldots, \tilde{x}_n - \Delta x_n)$. Since the values Δx_i are assumed to be small, we can expand this expression for Δy in a Taylor series, and neglect the terms that are quadratic (or of higher order) in Δx_i. As a result, we arrive at the following expression: $\Delta y \approx l$, where $l = f_{,1}\Delta x_1 + \cdots + f_{,n}\Delta x_n$, and $f_{,i}$ denotes the value of the partial derivative $\frac{\partial f}{\partial x_i}$ at the point $(\tilde{x}_1, \ldots, \tilde{x}_n)$.

If we want a guaranteed estimate, we must estimate the error of this approximation, i.e., we must find η such that $|\Delta y - l| \leq \eta$. The largest possible value of l is attained when each term $f_{,i}\Delta x_i$ in the sum attains its largest value. When $f_{,i} > 0$, this happens when Δx_i is the largest possible (i.e., when $\Delta x_i = \Delta_i$). When $f_{,i} < 0$, this happens when Δx_i takes its smallest possible value, i.e., when $\Delta x_i = -\Delta_i$. In both cases, this largest possible value of $f_{,i}\Delta x_i$ equals $|f_{,i}|\Delta_i$. Therefore, the largest possible value of the sum l is $L = |f_{,1}|\Delta_1 + \cdots + |f_{,n}|\Delta_n$.

Similarly, one can easily prove that the smallest possible value of l is $-L$. Therefore, possible values of l fill an interval $[-L, L]$. Since $|\Delta y - l| \leq \eta$, we can conclude that

$$|\Delta y| \leq |l| + |\Delta y - l| \leq L + \eta.$$

So, we can take $L + \eta$ as the desired estimate for Δ (in terms of intervals, we take

$$F = [\tilde{y} - (L + \eta), \tilde{y} + (L + \eta)].$$

(To complete the description of an algorithm, we need to specify how to compute the derivatives. If f is given as an analytical expression, then the derivatives can be computed automatically.)

This is a pretty good estimate for $Y = f(X_1, \ldots, X_n)$. The value L is a possible value of l for some Δx_i. For these Δx_i, we have $\Delta y = l + (\Delta y - l) \geq l - |\Delta y - l| \geq L - \eta$. Therefore, Δ (defined as a largest possible value of $|\Delta y|$) is also greater than or equal to $L - \eta$. So, $L - \eta \leq \Delta \leq L + \eta$.

When $\eta = 0$, we get $L + \eta = \Delta$, so $F = Y = f(X_1, \ldots, X_n)$. We are considering the case when quadratic terms are negligible. This means that terms that are quadratic in Δx_i (in particular, η, which is our estimate for these terms) are much smaller than terms that are linear in Δx_i (in particular, than l, and hence, than L which is the largest possible value of l). So, $\eta \ll L$, and therefore, $L + \eta$ is a pretty good estimate for $\Delta \in [L - \eta, L + \eta]$.

The above-described algorithm consists of two steps.

1. Estimating the derivatives $f_{,i}$.
2. Computing $L = |f_{,1}|\Delta_1 + \cdots + |f_{,n}|\Delta_n$, and $L + \eta$.

As soon as we know $f_{,i}$, the second step can be parallelized in exactly the same way as in the case of linear f. But the main time-consuming part of the algorithm is computing derivatives. So, if we really want to save computation time, we must parallelize the first part.

Such a parallelization has been proposed by Shiriaev in ([89]). There are two main ideas here.

- First, we can compute all n derivatives in parallel. In particular, if we have at least n processors, we can ask each of them to compute only one partial derivative.
- If we have more than n processors, then we can go further and parallelize the computation of each derivative.

The main idea of the second step is similar to the above-described idea of how almost every algorithm can be parallelized (with the main difference that there, this idea did not always work, while here, this idea always works). When we say that f is given as an analytical expression, we mean that f is represented as a kind of "computational scheme" that consists of several elementary computational steps. Each of these steps consists either in applying an elementary function to some already known value (that is either given or has already been computed), or in applying an arithmetic operation to two different numbers whose values are already known. This computational scheme is exactly how a compiler will perform these computations, if we write this analytical expression in any high-level programming language (FORTRAN, Pascal, C, etc).

For example, the expression $\sin(x^2 + 3 \cdot x)$ corresponds to the following computational scheme (in this scheme, r_i will denote the value that is computed on step i).

1. We start with a given value $r_0 = x$.
2. We apply a function x^2 to the known value $r_0 = x$, thus getting $r_1 = x^2$.
3. Then, we apply an arithmetic operation $\cdot$ (multiplication) to 3 and $r_0 = x$, thus getting $r_2 = 3 \cdot x$.

4. Third, we apply an arithmetic operation + to $r_1 = x^2$ and $r_2 = 3 \cdot x$, thus getting $r_3 = r_1 + r_2$ $(= x^2 + 3 \cdot x)$.

5. Finally, we apply sin to r_3, thus getting the final result r_4.

To describe how the above-described idea of Shiriaev works for a given function f, let's consider the final operation in this computational scheme. This operation can be either an arithmetic operation *op* that is applied to two given terms a and b, or an elementary function g applied to a given term a.

In the first situation, depending on exactly which arithmetic operation is used, we have the following well-known formulae for $f_{,i}$:

$$\begin{array}{lll} \text{if } f = a + b, & \text{then} & f_{,i} = a_{,i} + b_{,i}; \\ \text{if } f = a - b, & \text{then} & f_{,i} = a_{,i} - b_{,i}; \\ \text{if } f = a \cdot b, & \text{then} & f_{,i} = a_{,i} \cdot b + b_{,i} \cdot a; \\ \text{if } f = a/b, & \text{then} & f_{,i} = (a_{,i} \cdot b - b_{,i} \cdot a)/b^2. \end{array}$$

In the second situation, when $f(\vec{x}) = g\big(a(\vec{x})\big)$, we have the following formula:

$$f_{,i} = g'(a) \cdot a_{,i}.$$

In the first four cases, we can compute $a_{,i}$ and $b_{,i}$ in parallel (for multiplication and division, we can also compute a and b in parallel), and then compute $f_{,i}$. In the second situation (i.e., in the fifth case), we can compute $g'(a)$ and $a_{,i}$ in parallel.

If we have more than two processors, then we can similarly parallelize the processes of computing $a_{,i}$ and $b_{,i}$, etc.

5.3 Explicit parallelization of algorithms

5.3.1 Parallelizing interval computations

For all interval operations, we can halve the computation time if we run two processors in parallel according to the following ideas.

The standard interval computation techniques of computing an interval F from given f and X_i consists of the following: we describe the given

algorithm f as a sequence of elementary computational steps, i.e., arithmetic operations and applications of elementary functions, and then, we follow this description step-by-step, using operations with intervals instead of operations with real numbers.

The possibility of parallelization stems from the fact that interval operations are parallelizable: we can usually employ from two to four different processors to compute the lower and upper endpoints of the resulting interval. This fact can be easily traced for all arithmetic operations:

$$\begin{aligned} [a_1, a_2] + [b_1, b_2] &= [a_1 + b_1, a_2 + b_2] \\ [a_1, a_2] - [b_1, b_2] &= [a_1 - b_1, a_2 - b_2]. \end{aligned}$$

For $+$ and $-$, we can compute two endpoints $a_1 \pm b_1$ and $a_2 \pm b_2$ of the results in parallel.

$$[a_1, a_2] \cdot [b_1, b_2] = [\min(a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2), \max(a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2)].$$

Here, we can use 4 processors to compute the products $a_1 b_1$, $a_1 b_2$, $a_2 b_1$, and $a_2 b_2$, and then two of these processors to compute min and max of these four values.

$$\begin{aligned} 1/[a, b] &= [1/b, 1/a] && \text{if } 0 \notin [a, b], \\ X/Y &= X \cdot (1/Y), \\ g([a, b]) &= [g(a), g(b)] && \text{if } g \text{ is increasing, and} \\ g([a, b]) &= [g(b), g(a)] && \text{if } g \text{ is decreasing.} \end{aligned}$$

Operations X/Y and g are evidently parallelizable; X/Y is parallelizable because it is a sequence of two parallelizable operations.

This parallelization suffers from the necessity to have too many communication steps, so in general, it will not necessarily reduce the computation time.

In some cases, a similar idea does speed-up the computations.

- For integral equations, a similar parallelization idea was proposed in [65].
- For ordinary differential equations, a similar idea was used in [16] and [91].

This idea leads at best to halving the time, because we compute two endpoints in parallel instead of computing them sequentially.

In some cases, a similar idea can lead to a much better speedup. For example, we consider the solution of a system of interval linear equations $\sum_j a_{ij} x_j = b_i$ where, for the coefficient matrix a_{ij} and the right-hand side vector b_i, we know only intervals of possible values $A_{ij} = [a_{ij}^-, a_{ij}^+]$ and $B_i = [b_i^-, b_i^+]$. For different values $a_{ij} \in A_{ij}$ and $b_i \in B_i$, different solutions are possible. It is known that the right endpoint x_j^+ of the interval X_j of possible values of x_j is equal to the biggest value x_j among all the solution of the system $\sum_j a_{ij}^{\pm} x_j = b_i^{\pm}$ (that is obtained from the original one by substituting the endpoint instead of a_{ij} and b_i). Similarly, the lower endpoint x_j^- is equal to the smallest of these values.

These auxiliary systems can be solved in parallel, thus speeding up the computation. This algorithm can be further sped up if we take into consideration that not all combinations of endpoints are necessary to estimate $x_j^{\pm}$ ([15]).

5.3.2 Finding solutions in parallel

If we are solving an equation or a system of equations, then a natural idea is to divide the domain into several subdomains and use several parallel processors to check different subdomains for possible solutions. In particular, if this equation (or system of equations) has several solutions, and these solutions are sufficiently different (so that they belong to different subdomains), then each processor will compute its own root. For polynomial and for more general equations, such algorithms were presented in [1, 2, 7, 20–25, 29–31, 33, 34, 66, 78–84, 87 (Section 4.3.3), 88, 90, 94, 97, 98, 104-110].

For particular examples we have:

- parallel Newton's method is described in [66] (see also [94]);
- parallel chord method for finding roots is described in [88];
- parallel bisection method is described in [90].

We presented these techniques in this order because: Newton's method is known to be the fastest, the chord method is not as fast, and the bisection

method is the slowest of the three. Therefore, whenever possible, Newton's method is preferable.

However, Newton's method is not always applicable: it requires that the function be smooth (differentiable), and that we can compute the derivatives. The chord method is also convergent only under some conditions. The bisection method is the most universal: it is applicable whenever we have a function $f : [a, b] \to R$ for which the values $f(a)$ and $f(b)$ in endpoints a and b have different signs.

[90] describes applications of parallel interval computations of the roots to chemical and petroleum engineering.

[23–25, 83] use *circular complex arithmetic* instead of the regular interval computations (i.e., a formalism in which basic objects are not intervals, but circles in a complex plane).

5.3.3 Matrix operations

Another example of explicit parallelization is matrix multiplication $c_{ij} = \sum_k a_{ik} b_{kj}$: computation of different elements c_{ij} can be done in parallel (see, e.g., [13]).

5.3.4 Non-linear functions

For essentially non-linear functions f, no generally applicable parallelization is known. However, in many cases, parallelization can be naturally extracted from the nature of the problem. In the following three subsubsections, we will describe three such cases.

5.3.5 Artificial Intelligence (AI)

Crudely speaking, the main idea of Artificial Intelligence (not the best name, but still used for historical reasons) is as follows. To use a computer to solve a problem, we must describe an algorithm. It would be nice if we could just describe a *problem*, i.e., describe what we know and what we want to compute, and *the computer will choose* or design *an* appropriate *algorithm and solve this problem.* In other words, we want to find a "meta"-algorithm, that would take an arbitrary problem and solve it.

In general, this problem is undecidable in the sense that an algorithm that solves all the problems is impossible. However, there exist systems that have already solved many important problems (e.g., expert systems).

To make an AI system work, we must fill it with knowledge. This knowledge mainly comes from measurements. For many of these measurements, we do not know the probabilities of different values of measurement error. Therefore, for these measurements, the only information that we have about an error is its upper bound Δ. Hence, if we measured the value x and the result is $\tilde{x}$, then the only information that we have about the actual value of x is that x belongs to an interval $[\tilde{x} - \Delta, \tilde{x} + \delta]$. In view of this remark, it is no wonder that intervals are used to represent knowledge in AI.

Another reason why intervals are used in AI is that when we represent knowledge in an expert system, we must not only describe everything that we know, but we must also enable the computer to distinguish between the absolutely reliable knowledge and heuristic rules, i.e., rules that in general works fine but can sometimes err. In other words, we need to somehow describe that our degree of belief in some statement is larger than our degree of belief in some other rule or fact.

In standard mathematical logic, a statement is either true or false, typically represented by 1 and 0, respectively. In other words, absolute belief in a statement A is represented by the number 1 and absolute belief that A is false is described by the number 0. Therefore, it is natural to represent the intermediate degrees of belief $t(A)$ in different statements A by numbers between 0 and 1. This natural idea is used in the majority of expert systems.

How to find the value $t(A)$? There are many methods. For example, we can fix a sequence of "standard" statements, and establish a degree of belief in a given statement A by comparing this degree of belief with the degrees of belief in these standard statements. However all methods can generate only finitely many bits of $t(A)$. Therefore, they cannot generate a precise value of $t(A)$; they give only an *approximate* value. If we ask an expert more questions, we may get a better idea of her degrees of belief, but after each sequence of questions, we have only intervals of possible values of $t(A)$.

For example, suppose that we have fixed 11 different statements $A_0, \ldots, A_{10}$ as standard ones, and we have defined their degrees of belief as $0, 0.1, \ldots, 0.9, 1.0$. If we know that our degree of belief in some statement A is larger than in A_5 but smaller than in A_6, then the only thing that we know about $t(A)$ is that $0.5 < t(A) < 0.6$. In other words, the only thing we know about $t(A)$ is that $t(A) \in [0.5, 0.6]$.

Let's describe one more source of interval uncertainty in AI. Suppose that out database contains two statements A and B, with degrees of belief correspondingly $t(A)$ and $t(B)$. We ask a query "$A\&B$?" If A and B are both absolutely true, then we can also be sure that $A\&B$ is true, so the answer to this query is "yes".

If we are not absolutely sure in A and B, i.e., if $t(A) < 1$ and $t(B) < 1$, then we are not absolutely sure that $A\&B$ is true. So, as an answer to this query, we must (if we want this system truly to behave like an expert) to convey a (reasonable) degree of belief in $A\&B$.

If a complex statement $A\&B$ is in our database, with a degree of belief attached, then we just return this degree of belief as an answer. But what if it is not? From the commonsense viewpoint, the problem is ambiguous, because there are several possible cases.

- It could be that arguments in favor of A and B come from independent sources. Here, $t(A)$ is our degree of belief in the first source, and $t(B)$ is our degree of belief in the second source. Since $t(A) < 1$ and $t(B) < 1$, we have doubts in both sources. The only case when we conclude $A\&B$ is when we believe in both sources. Even if A is true, B can still be false. Therefore, our degree of belief in $A\&B$ is smaller than $t(A)$ and $t(B)$.
- It could also be that arguments in favor of A and B come from the same source. So, if we believe in A, we thus believe in B and hence, we believe in $A\&B$. In this case, $t(A\&B) = t(A)$.

Since in practical expert systems, we do not keep track of the arguments (that would require enormous memory), all we keep is the values $t(A)$ and $t(B)$. So, for $t(A\&B)$, we can thus have different values.

If two real numbers $a < b$ are possible values of $t(A\&B)$, then we can imagine all kind of intermediate situations, in which $t(A\&B)$ will take all

the values intermediate between a and b. As a result, for given $t(A)$ and $t(B)$, the set of all possible values of $t(A\&B)$ contains all intermediate points. Therefore, it is an *interval.*

So, even if the degrees of belief in basic statements are precise, the degree of belief in combined statements is best described by an interval. (See, e.g., [41–43]).

So, the knowledge in the knowledge base has interval uncertainty, and the degrees of belief are also described as intervals. Therefore, if we ask this expert system about the value of some quantity, most probably the answer will also be an interval, and not a value. To what extent is this answer reliable? This is described by the degree of belief in this answer. This degree of belief is computed based on the initial degrees of belief (of statements from the knowledge base). These initial degrees of belief are not known precisely: we only know intervals that contain them. Therefore, the resulting degree of belief will also not be known precisely: the only thing that we can really generate is the interval of possible values of reliability. So, when we ask a query, actually computations with intervals are going on in the expert system. In other words, such expert systems actually perform interval computations.

We have already mentioned that AI systems are aimed at solving general problems that cannot be easily handled by the existing methods. No wonder that the algorithms of AI are often very time-consuming. If we use intervals instead of numbers, the computational complexity increases, and the situation becomes even worse. So, parallelism is badly needed.

Fortunately, in the majority of the cases, there is a natural parallelization: namely, our knowledge is usually (more or less) compartmentalized. This means that when we look for a solution for a mathematical problem, we know for sure that our knowledge of, e.g., ethics will not help. So, for any given query, we do not need to look into all possible rules and facts from the knowledge base: only into those that are relevant. So, if we have several queries that are relevant to different parts of the knowledge base, then in principle, we can handle them in parallel.

Even if we have one query, we can still obtain a speedup if we try to relate this query with different parts of the knowledge base (this can be done in parallel). If in an attempt to answer the query, the computer will generate several auxiliary queries, they can also be answered in parallel.

Such natural parallelizations are described, e.g., in [54] and [99].

We look at two examples:

1. For *automated manufacturing*, the object to control consists of several instruments on the shop floor. These instruments are reasonably independent, so it is possible to control them by assigning a processor to each of them, and reducing the communication between these processors to the necessary minimum. The input comes from sensors, so actually, for each variable, we know the *interval* of possible values. Therefore, the rules that form a decision-making expert system are also formulated in terms of intervals. Corresponding parallelization algorithms are described in [59].

2. For a general purpose *medical system*, the area of knowledge is naturally divided into several subareas (corresponding to different medical specializations), so here, a natural parallelization can be used. This parallelization is presented in [41–43].

5.3.6 Spatially localized objects

In many problems, we are analyzing objects that are located in real (3–D) space. The properties of real-life objects are often *local* in the following sense: what happens to an object that is located at a point $\vec{x}$, depends mainly on the situation in a neighborhood of $\vec{x}$. Therefore, if we have several spatially separated objects, we can process the evolution of each object in parallel (with only a little bit of communication required).

This idea can be used to solve partial differential equation (see, e.g., [85]). For example, if as part of this solution, we must compute an integral, we can use several processors to compute the integral over several subdomains, and then add up the results.

This idea does not always work because there are physical situations when action-at-a-distance has to be taken into consideration (e.g., in celestial mechanics), but it works in many real-life situations.

Another area where this idea is helpful is *computer graphics* [44, 45, 67]. If we wish to generate an image that consists of several objects, we must include:

- how the objects interact,
- what is the intensity at different points,
- what is visible and what is not,
- how the objects cast shadows on one another,
- etc.

All of these properties are parallelizable, e.g., to find an interaction of two sizeable bodies, it is sufficient to divide the entire 3–D (or 2–D) domain into several subdomains, and describe the interaction in each subdomain. We can then employ several processors to handle each subdomain. To speed up interval methods, the subdivision is done not by halving, but at the critical points of the curves and surfaces.

5.3.7 Monte-Carlo methods

In Monte-Carlo methods, we perform several simulations, and then process the results $z^1, \ldots, z^N$ of these simulations. These simulations are independent, and therefore, they can execute on different processors in parallel (see, e.g., [5, 12, 46–48]). In particular, in [12], this technique is applied to AI problems (namely, to the problem of finding an interval of possible values of degree of belief).

5.4 Parallelizing interval computation in synthesis problems

We have already mentioned that there are two types of synthesis problems:

- problems that require solving a system of equations (possibly including inequalities), and
- optimization problems.

In the following subsections, we will describe how to parallelize the corresponding interval computations.

5.4.1 Solving a system of equations and inequalities in parallel

In the previous subsections, we described how to parallelize interval algorithms that solve a system of equations. Let us describe how to parallelize the solution of a system of equations and inequalities. This idea was described in [3, 74, 75].

Suppose that we have a system of equations and inequalities. Initially, to each variable, let us assign an interval $[-U, U]$ (here, U denotes the largest real number that can be represented in the particular computer). At each iterative step, for each variable x_i, we decrease the interval for each variable x_i. The idea behind this iteration is as follows: suppose that we are dealing with a variable x_i (e.g., x_1), and we have an equation (e.g, $x_1 x_2 = 1$), or an inequality (e.g., $x_1 \le 0.3 x_2$) that contains this variable. Since we know the intervals that contain all possible values of all other variables in the equation (e.g., x_2) we can use this information to restrict the values of x_1.

For example, if we know that $x_2 \in [1, 2]$, then, from the equation $x_1 = 1/x_2$, we conclude that $x_1 \in [0.5, 1]$, and from the inequality $x_1 \le 0.3 x_2$, that $x_1 \in (-\infty, 0.6]$. Combining these two estimates, we conclude that $x_1 \in [0.5, 0.6]$.

These iterations can be easily parallelized because at each iteration, computation for each equation and for each variable can be done independently (on different processors). In order to obtain a new interval for each variable x_j, it remains only to take the intersection of the intervals obtained from the different equations and inequalities that contain x_j.

This intersection, in its turn, can also be parallelized. Indeed, what is an intersection from a computational viewpoint? The intersection of the intervals $[x_\alpha^-, x_\alpha^+]$ is an interval $[x^-, x^+]$ with $x^+ = \min x_\alpha^+$ and $x^- = \max x_\alpha^-$. Computing the minimum m of n numbers can be done in parallel in exactly the same way as we compute the sum of n numbers in parallel:

1) divide the set of numbers into two halves;

2) the first processor computes the minimum m_1 of the first half; at the same time, the second processor computes the minimum m_2 of the second half;

3) compute $m = \min(m_1, m_2)$.

Similarly, the maximum can be computed in parallel.

5.4.2 Parallel interval optimization

The main idea of parallelizing interval algorithms for solving optimization problems is relatively simple. Suppose that we are given a function f defined on some area $\Omega \subseteq R^n$, and we must find a value $\vec{x}$ at which f attains its maximum. Suppose also that we have two processors that can work in parallel. Then, we can speedup our computations as follows: divide the area Ω into two subareas Ω_1 and Ω_2, find the maximum of f on each of them, and then compare the results. If the first processor finishes its computations faster than the second one, then we can again divide the remaining area between the two processors. If we have more than two processors, then we can obtain an even greater speedup.

This idea has been efficiently applied in [8, 15, 32, 56, 57]. In particular it is applied to finding local extrema: for functions of one variable (in [103]) and in the general case [60].

5.5 Implicit parallelism

Up to now, we have described special parallelization techniques for interval computations. If none of them is applicable, we can try to apply general parallelization techniques to the particular case of interval computations.

5.5.1 Applying general parallelization techniques

From the computer viewpoint, an interval is an *ordered pair* of real numbers. So, every computer methodology that parallelizes operations with *ordered finite sequences* can be helpful here. This idea is brought to the extreme in the works of Cooke, who proposed an ordered finite sequence (he calls it a *bag*) as the basic data type of his new high-level computer language BagL. Parallelization ideas of BagL are thus directly applicable to interval computations [10, 11].

5.5.2 Interval methods help to find what is parallelizable

Suppose that we cannot simply look at the algorithm and see that it is parallelizable, i.e., it has several independent parts, and the known methods of parallelization do not work. Looking for parallelizable parts is hard, but routine, work. So maybe we can *use the computer itself to parallelize the algorithm?* A computer understands only a very formal language. So, to involve a computer, we must formulate the problem of finding independent parts as a mathematical problem, and solve this problem (by using either known mathematical techniques or some specially designed method).

At first glance, the problem is extremely complicated. Programs can be of arbitrary complexity, and analyzing them is tough. However, there is a way to overcome this difficulty: A program written in a high-level language consists of several straightforward computations and loops. It is reasonably easy to check interdependency of straightforward computational steps. However, these steps do not represent a big time problem, because they are implemented only once. The main time-consuming part of an algorithm is a loop. So, in order to really save time when parallelizing an algorithm, we must be able to answer the following question: is there data dependency in a loop, or a loop can be safely parallelized? In [112], this question is reformulated as a mathematical problem, and this mathematical problem is solved by interval methods.

5.5.3 Interval methods help to parallelize

When we discover that an algorithm is parallelizable, then we must *actually parallelize* it, i.e., distribute the parallelizable jobs between the processors. If we know the job lengths, then we can formulate this scheduling as a precise mathematical problem. However, in real life, the job lengths are not precisely known: we have only intervals of possible values.

In [58, 77], algorithms are given that schedule the jobs under this interval uncertainty.

A different approach to dynamic load balancing is given in [15].

5.5.4 We can save even more time if we do not start computations with full accuracy

In the above text, we counted the computation time in terms of elementary operations. In reality, the time that is required for an elementary operation, depends on the *accuracy* required. Adding two double-precision real numbers takes more time than adding two regular real numbers. Usually, if we are interested in accurate results, we (out of caution) start with the maximal possible precision. In many cases, such precision is unnecessary, so we waste time.

The same problem appears when we choose a precision for auxiliary problems that appear as part of our algorithm. For example, if we must solve a system of linear equations as an intermediate step, then we usually solve it with the maximum precision possible, thus using much more iterations than necessary.

This makes sense for sequential computations, because who knows, maybe for this particular problem maximum precision is necessary. But in parallel computations, we are no longer required to do that. This idea was proposed by Yakovlev in [114] and further developed and implemented by Musaev ([68–72]):

1) we can start the computations with a reasonable precision (if we have enough processors at our disposal, we can at the same time start computations with a better precision);

2) if at some point, the resulting precision is not sufficient, i.e., if the intervals are too wide, then we will redo all the computations that led to this particular point (or, if have already started computations with a better precision, follow them further).

This way, we only redo the computations that influenced our results.

Computations in general can be compared with a wave: we pass data from one processor, from one process, to another. In terms of this analogy, we (if necessary) reverse the wave and start it anew. Because of this comparison, methods that involve such processor communication are called *wave computations* [114].

5.6 Non-parallelizable algorithms & neural networks

If an algorithm that we are trying to parallelize *is hard to make parallel* (or if we do not have an algorithm in the first place), then we can follow the idea that we outlined earlier: stop trying to parallelize the algorithm and *try to find another (parallelizable) algorithm that computes the same function* $y = f(x_1, \ldots, x_n)$.

Since initially, we could not design a parallelizable algorithm that computes a given function f, this design is clearly not easy to find. Therefore, we cannot rely on our abilities and/or intuition to search for such an algorithm. We must use a computer to help us design such an algorithm. A computer needs a very formal description of what is required. Therefore, we must formulate our design problem in mathematical terms. What does "parallelizable" means in these terms?

We begin by describing what "non-parallelizable" means. If we have a sequence of computational steps, and one of them can be started before the previous steps are done, then we can implement this step on a separate processor in parallel with the previous steps. So, "non-parallelizable" means that we have a long sequence of operations, and each operation cannot be started before we are done with all the previous ones.

Since we are designing an entirely new algorithm, we will make it as parallelizable as possible. This means that the designed algorithm will not have any long non-parallelizable sequences. This algorithm must therefore consist of a short sequence of simple "subroutines" (blocks) that pass their results to each other, and that can easily work in parallel.

What are these blocks? The simpler the faster, and thus the better. The simplest possible algorithms that deal with real numbers are linear functions (not only they are easy to compute, but, as we have seen, they are also easily parallelizable). We cannot, however, take all our blocks to be linear functions, because then the resulting parallel algorithm will compute compositions of linear functions, and these compositions are always linear. So, we need at least one non-linear block g if we want to compute a non-linear function f.

As a non-linear block, we can have non-linear functions of one, two, etc., variables. The more variables, the longer it usually takes to compute

a function. Therefore, the simplest block is when we take a function of one variable $g(x)$.

Because we want to save on the computation time, we would like to choose functions $g(x)$ whose computation take the smallest time. Usually, computing different non-linear functions takes varying computational time. Therefore, it is reasonable to expect that there is only one non-linear function $g(x)$ whose computation takes the smallest possible time. In this case, all the non-linear blocks will compute this function and will therefore be identical.

This "uniqueness" is more a guess than a theoretical conclusion. Hence, it is quite possible that there is an entire family of different functions $g(x)$ with the same computational time. In this case, the only conclusion that we can make is that all the nonlinear functions belong to the same family (and are in this sense similar).

The previous analysis leads to the following conclusion: we must design our algorithm from building blocks of the following two types:

1) blocks that implement linear functions, i.e., that transform input $x_1, \ldots, x_n$ into output $y = a_0 + a_1 x_1 + \cdots + a_n x_n$;

2) non-linear blocks, i.e., that transform input x into output $y = g(x)$ (where g is a non-linear function).

The computation time of a parallel algorithm is equal to the sum o the running times of its consequent components. Therefore, the total com putation time is minimized if we have the smallest number of consequen components. The fewer consequent components we have, the fewer com munications that are needed, and, therefore, the communication time i also minimized if we have fewer consequent components.

Among the building blocks that we just described, non-linear ones tak more time. So, the fewer of them, the faster. Since we cannot do witho them, we include at least one. If we had only several non-linear componen working in parallel, then we would never be able to compute anythi except functions of one variable. So, we also need some linear blocks well.

Let's show that we must reserve time for at least two linear block Indeed, suppose that the total computation time consists of the time

one linear block plus the time of one non-linear block. The linear block is either before the non-linear one or after it. In both cases only a limited set of functions can be computed.

- If the linear part is *before* the non-linear part, then the only functions that we can compute this way are $f(x_1, \ldots, x_n) = g(x_0 + c_1 x_1 + \cdots + c_n x_n)$. These functions have the property that their level surfaces $f(x_1, \ldots, x_n) = \text{const}$ are (hyper-)planes in an n-dimensional space. For many important functions of several variables (even for $f(x_1, x_2) = x_1 x_2$), this is not true.
- If the linear part is *after* the non-linear part, then the only functions that can be computed this way are $f(x_1, \ldots, x_n) = c_0 + c_1 g_1(x_1) + \cdots + c_n g_n(x_n)$. Each of these functions has the following property: its partial derivatives $\partial^2 f / \partial x_i \partial x_j = 0$ if $i \neq j$. Many important functions (including the same function $f(x_1, x_2) = x_1 x_2$) do not satisfy this property.

Therefore we require at least two linear steps.

We have just concluded that we need the time for at least three consequent blocks: a non-linear step and two linear steps. In what order will these components be implemented? In general, there are three possibilities:

1) both linear steps are before the non-linear one;
2) both linear steps are after a non-linear one;
3) one of the linear steps is before the non-linear one and the other is after it.

The first two possibilities are inadequate for our purpose.

1. If both linear steps are before a non-linear one, then we can combine these two linear steps into one linear step (since composition of two linear functions is again a linear function), and we have already shown that such algorithms are not universal.
2. Similarly, if both linear steps follow a non-linear step, then we can combine these two linear steps into one, and end up with a non-universal configuration.

Therefore one linear step must be before the non-linear one and one after

Finally, we arrive at the following configuration:

1) input n numbers $x_1, \ldots, x_n$;
2) K linear elements generate the values $y_k = w_{k1}x_1 + \cdots + w_{kn}x_n + w_{0n}$ $1 \le k \le K$;
3) compute the values $z_k = g(y_k)$, where g is a given non-linear functio (or $z_k = g_k(y_k)$, where all g_k belong to a given family of non-linea functions);
4) compute a linear combination of these values z_k: $y = W_0 + W_1 z_1 + \cdots + W_K z_K$.

As a result, we compute the following function:

$$y = W_0 + \sum_{k=1}^{K} W_k g\left(\sum_{i=1}^{n} w_{ki}x_i + w_{k0}\right).$$

This configuration is exactly a *3-layer neural network.*

Our justification of the above-given three-layer configuration was tha fewer layers were not sufficient to represent an arbitrary function. Horni et al in [35] answer the sufficiency question for a 3-layer network by provin that an arbitrary continuous function f defined on an n-dimensional cub can be, for every $\varepsilon > 0$, ε-approximated by an appropriate neural networ

(What we described is a *mainstream* model of neural networks. In man cases, more complicated model are used.)

The results from [35] assume that the function g computed by a no linear component of a network is precisely known. In real life, whethe we compute it on a hardware device, or in software, we obtain only a approximation to g. In other words, instead of a function g, we know a interval function $G(x) = [g^-(x), g^+(x)]$, and the only thing we know abou the actual g is that $g(x) \in G(x)$ for all x. Can we still find a neural netwo design that will guarantee that we approximate a given function f with given accuracy ε?

Of course, if this interval is large, and the desired accuracy ε is sma then we cannot guarantee this accuracy. So, the question is: is it possib

for every ε, to find the design *and* the necessary accuracy of the component neurons, i.e., accuracy δ with which we can compute g, that will guarantee computing f with accuracy ε?

The intuitive answer that comes to mind is "yes" (particular for those familiar with approximation theory); this answer is proven in [92, 96].

Stated more formally: if we know a function f that relates $x_1, \ldots, x_n$, and y, then for every $\varepsilon > 0$, there exists a neural network that computes a function $\tilde{f}(x_1, \ldots, x_n)$ with a property that

$$|f(x_1, \ldots, x_n) - \tilde{f}(x_1, \ldots, x_n)| \leq \varepsilon.$$

Since the main problem that we are dealing with was the problem of processing measurement results, we are talking about approximate measurements, and so, if ε is small, it is quite sufficient to have a function that is ε-close to f.

This result proves that there exists such a neural network, but it does not give an algorithm for constructing this network. Such an algorithm was first described in [46, 53]. A modification of this algorithm is presented in [63] and [73].

5.6.1 Neural computations & intervals

Suppose that we have already implemented an algorithm $f(x_1, \ldots, x_n)$ as a neural network and desire to solve the basic problem of interval computations for this algorithm: If we know the intervals X_i of possible values of x_i, then what is the interval Y of possible values of y? This problem is described in [27].

Two solution strategies are:

1. Apply a method that is similar to the standard idea of interval computations: follow the algorithm, and for all its components, i.e., for linear combinations, and for $g(x)$, compute the interval of possible values of the result based on the input intervals.

   Thus we obtain an interval $F \supseteq Y$ with no big increase in computation time. However, the resulting interval F may be much wider that Y.

2. If the intervals are sufficiently narrow (so that terms quadratic in Δx_i, can be neglected), there is another possibility: to apply a Monte-Carlo style method [46–48, 50]. This method requires that we run several processes of computing f in parallel, and then process the results. So, we need to duplicate the original neural network as many times as we need to repeat the computations, and then add one more neural network to process the results. In comparison with the first idea, we spend more computation time and use more processors. But as a result, we get the exact value of Y (or, to be more precise, the value of Y that is accurate if we can neglect terms quadratic in Δx_i).

Several *applicatons of these ideas have been presented.*

- The general idea of how to apply neural networks to controlling plants with interval uncertainty is described in [55]:

  1) train a neural network to simulate an object;

  2) apply minimization techniques to find the optimal control.

- [55] describes the problem of stabilizing vibrations in large space structures.

- [76] describes a problem in radiation therapy: we have 360 pencil beams. We must select the intensities so that the effect is desirable. Neural network is used to learn how to do that.

6 Improving interval estimates with parallelism

The basic problem of interval computations, given an algorithm f and intervals $X_1, \ldots, X_n$, is to find an interval $F \supseteq f(X_1, \ldots, X_n)$. The traditional methodology to solve this problem consists of following an algorithm f step-by-step, but using on each step operations with intervals instead of operations with numbers. This methodology leads to overestimated intervals F because:

1. The methodology does not take into consideration that we the intervals that we obtained at the intermediate steps of an algorithm f, may not be independent. For example, if $f = x - x$ and $X = [0, 1]$, we subtract [0,1] from [0,1] using the formulae of interval arithmetic. The result is $F = [-1, 1]$, while the correct answer $f(X) = [0, 0]$.

2. If an algorithm has a branch, i.e., an if-then statement, then the set of possible results is not an interval but a union of several disjoint intervals that correspond to different cases. For example, if our algorithm computes the square root of x_1 and then "if $x_2 > 0$ then $f := \sqrt{x_1}$ else $f := -\sqrt{x_1}$", then for $X_1 = [1, 2]$ and $X_2 = [-1, 1]$, we have $f(X_1, X_2) = [-\sqrt{2}, -1] \cup [1, \sqrt{2}]$, while the smallest interval that contains this set ($[-\sqrt{2}, \sqrt{2}]$) contains many extra elements.

In the next two subsections, we will describe methods that have been proposed to overcome these problems and we will show how to parallelize these methods.

6.1 Hansen's method

A method to overcome the first problem was proposed by Hansen in [28]. The main idea of Hansen's method is that at each intermediate computational step, we keep track of the dependency of the result of this step on the input values $x_1, \ldots, x_n$.

To be more precise, we assume that we have n intervals $X_1, \ldots, X_n$, and a function f, and we are interested in estimating $f(X_1, \ldots, X_n)$. At each step, instead of intervals, we are dealing with "generalized intervals" of the type $A = (\tilde{a}, a_1, \ldots, a_n, A_q)$, where $\tilde{a}$ and a_i are real numbers, and A_q is a non-negative real number (that is supposed to bound the terms that are quadratic in Δx_i). This interval means that for arbitrary values $x_i \in X_i$, the value of the auxiliary variable a is equal to $\tilde{a} + a_1 \Delta x_1 + \cdots + a_n \Delta x_n + q_a$, where $|q_a| \leq A_q$, $\Delta x_i = \tilde{x}_i - x_i$, and $\tilde{x}_i$ is the midpoint of X_i (this representation coincides with linear order terms in Taylor series; a_i is a partial derivative of a with respect to x_i).

In particular, since $x_i = \tilde{x}_i - \Delta x_i$, we thus represent x_i as

$$\big(\tilde{x}_i, 0, \ldots, 0, -1_{(\textit{at the i-th place})}, 0, \ldots, 0, 0\big).$$

Arithmetic operations are defined in a natural way.

- If $a = \tilde{a} + a_1\Delta x_1 + \cdots + a_n\Delta x_n + q_a$, $|q_a| \le A_q$, and $b = \tilde{b} + b_1\Delta x_1 + \cdots + b_n\Delta x_n + q_b$, $|q_b| \le B_q$, then $a + b = (\tilde{a}+\tilde{b}) + (a_1+b_1)\Delta x_1 + \cdots + (a_n+b_n)\Delta x_n + (q_a+q_b)$. Since $|q_a+q_b| \le |q_a| + |q_b| \le A_q + B_q$, we can define addition of generalized intervals as $(\tilde{a}, \vec{a}, A_q) + (\tilde{b}, \vec{b}, B_q) = (\tilde{a}+\tilde{b}, \vec{a}+\vec{b}, A_q+B_q)$.

- A similar formula works for subtraction:

$$(\tilde{a}, \vec{a}, A_q) - (\tilde{b}, \vec{b}, B_q) = (\tilde{a}-\tilde{b}, \vec{a}-\vec{b}, A_q-B_q).$$

- For multiplication, the formula is slightly more complicated:

$$\begin{aligned} ab &= (\tilde{a} + a_1\Delta x_1 + \cdots + a_n\Delta x_n + q_a)(\tilde{b} + b_1\Delta x_1 + \cdots + b_n\Delta x_n + q \\ &= (\tilde{a} + \vec{a}\cdot\vec{\Delta}x + q_a)(\tilde{b} + \vec{b}\cdot\vec{\Delta}x + q_b) = \tilde{a}\tilde{b} + (\tilde{a}\vec{b} + \tilde{b}\vec{a})\cdot\vec{\Delta}x + q \end{aligned}$$

where $q = (\vec{a}\cdot\vec{\Delta}x + q_a)(\vec{b}\cdot\vec{\Delta}x + q_b) + (\tilde{a}q_b + \tilde{b}q_a)$. Since $|\Delta x_i| \le \Delta_i$ we can conclude that

$$|\vec{a}\cdot\vec{\Delta}x| = |a_1\Delta x_1 + \cdots + a_n\Delta x_n| \le |a_1|\Delta_1 + \cdots + |a_n|\Delta_n.$$

Likewise, $|\vec{b}\cdot\vec{\Delta}x| \le |b_1|\Delta_1 + \cdots + |b_n|\Delta_n$. Therefore,

$$\begin{aligned} |q| \le\ & (|a_1|\Delta_1 + \cdots + |a_n|\Delta_n + A_q)(|b_1|\Delta_1 + \cdots + |b_n|\Delta_n + B_q) \\ & + |\tilde{a}|A_q + |\tilde{b}|B_q \end{aligned}$$

and we can define multiplication as

$$AB = (\tilde{a}, \vec{a}, A_q)(\tilde{b}, \vec{b}, B_q) = \left(\tilde{a}\tilde{b}, \tilde{a}\vec{b} + \tilde{b}\vec{a}, (AB)_q\right)$$

where $(AB)_q$ denotes the following expression:

$$(|a_1|\Delta_1 + \cdots + |a_n|\Delta_n + A_q)(|b_1|\Delta_1 + \cdots + |b_n|\Delta_n + B_q) + |\tilde{a}|A_q + |\tilde{b}|B_q$$

- Similar estimates are obtained for division, and for the application of elementary functions.

In the method as described, quadratic terms are bounded from above by A_q and from below by $-A_q$. There exists a version of this method in which two different numbers are used for the lower and upper bounds.

Hansen's method leads to much better estimates for F than the traditional interval techniques. However the method takes excessive computational time: for every operation, we need at least $n+1$ operations with numbers instead of one. For large n, this is significant.

Addition and subtraction are defined component-wise, so we can use $n+2$ processors to make these additions and subtraction *in parallel.* For multiplication $C = AB$, computing $\tilde{c}$ and $\vec{c}$ is also done component-wise (and is thus *parallelizable*). Computing C_q consists of computing two linear combinations, and we already know how to *parallelize* such computations. Since a vector part of the generalized interval contains partial derivatives, the parallelization of this part is done in exactly the same manner as the parallelization of automated differentiation.

6.2 Multi-interval computations and their parallelization

If we wish to avoid the second problem, then, instead of an interval, we must consider finite unions of intervals $\mathcal{I} = \cup I_i$. Such unions are termed *multi-intervals.* Arithmetic operations *op* and application of elementary functions g are defined for multi-intervals $\mathcal{I}$ and $\mathcal{J}$ in the same manner as for regular intervals: $op(\mathcal{I}, \mathcal{J}) = \{\, op(x,y) \mid x \in \mathcal{I}, y \in \mathcal{J}\}$ and $g(\mathcal{I}) = \{g(x) \mid x \in \mathcal{I}\}$.

If $\mathcal{I} = \cup I_i$ and $\mathcal{J} = \cup J_j$, then in terms of component intervals I_i and J_j, these operations take the following form:

- $g(\mathcal{I}) = \cup g(I_i)$; the result is a multi-interval with component intervals $g(I_i)$; For different i, these intervals can be computed in parallel;
- $op(\mathcal{I}, \mathcal{J}) = \cup op(I_i, J_j)$; the result is a multi-interval with components $op(I_i, J_j)$.

The idea of computing multi-intervals in paralleal is described in [113] (see also [93]), where it is shown that each elementary step of multi-interval computation can be computed in parallel.

- For $g(\mathcal{I})$, component intervals I_i for different i can be computed in parallel.
- For $op(\mathcal{I}, \mathcal{J})$, component intervals $op(I_i, J_j)$ for each pair (i, j) can be computed in parallel.

6.3 Multiple methods and their parallelization

We have already mentioned that if we compute an estimate F for $f(X_1, \ldots, X_n)$ using different techniques, and then take the intersection, this will often result in a better estimate F. By "using different techniques", we mean one of the following options:

- applying one and the same method (e.g., naive interval computations) to different expressions that represent the same function f (e.g., $x_1(x_2 + x_3)$ as opposed to $x_1x_2 + x_1x_3$); or
- applying different version of a method (e.g., different interval iterative algorithms) to one and the same expression; or
- applying different methods (e.g., naive interval computations and Hansen's method) to one and the same expression; or
- applying different methdos to different expressions.

Since all these methods can be run in parallel, we do not lose any computational time. An important particular case of this idea ([118]) is when parallel computers actually compute different sets containing the desired results: e.g., one computer computes a rectangle that contains the desired pair (x_1, x_2), another computer computes a circle, the third one uses ellipsoid arithmetic to compute an ellipsoid that contains (x_1, x_2), and the fourth may compute an analytical expression for the result. These parallel processes represent, so to say, different *aspects* of localization, so their simultaneous computation is called *multi-aspectness* in ([118]).

Some of these methods are *iterative*; for example, some interval computation techniques for finding a root x of a function lead to a sequence of decreasing intervals each of which contain the desired root. On each step, we use the previously computed interval to compute the updated (smaller)

one. If we have several iterative algorithms working in parallel, then as soon as one of them computes a new iteration (i.e., a new *locus* for x), it can forward this locus to other algorithms, and these other algorithms can use this locus in their own computations (this idea is called *recomputation*). Thus, we have another example of wave computations. These (and similar) ideas were described in [114–118] (they can be used in conjunction with the above-described case of wave computations).

It is not necessary to wait until the very end of computation to calculate this intersection. Suppose, e.g., that we are solving a differential equation (or a system of such equations). The majority of methods for solving these equations consist of computing the values of the unknown functions $\vec{x}(t_0)$, $\vec{x}(t_0+\Delta t), \ldots$, in consequent moments of time. If we use different methods with the same integration step Δt, then after each step, we can intersect the intervals obtained by different methods and thus obtain a better interval estimate for $x(t)$. This idea allows us to avoid the "wrapping effect" for differential equations (see, e.g., [62, 116, 117]).

This idea can be also used as a *test for* the *consistency* of the original problem (if the intersection is empty, then this problem is inconsistent), and if we know that the problem is consistent, as an additional *test for errors* (if we know that a problem is consistent, and the intersection is empty, that means that we have made a mistake somewhere).

7 Hardware

We have already mentioned that the efficiency of a parallelization depends essentially on how many communication steps we need. Since a communication step usually takes much more time than a computation step, each parallelization that creates new communication steps runs the risk that the communication time will exceed the expected decrease in computation time, and therefore, the resulting algorithm will not be efficient. So, the problem of choosing an approproate hardware is crucial for the success of a parallel algorithm.

In this section, we will briefly describe the existing choices, both for the general case and for specific parallel interval algorithms. These specific

architectures will be described in the same order in which the corresponding algorithms were described in the text.

7.1 General case

For a general case, the possibility of using transputers for interval computations is described in [13]. More precisely, this paper deals with the possibility of adding the possibility of parallel computation on transputers to the interval modification of Pascal, PASCAL–XSC.

7.2 Interval linear operations

One can use transputers and still gain a reasonable speedup ([13, 100]). Another possibility is to use a more specialized hardware.

When we are solving interval linear problems, the main computational operations are linear operations with real numbers. Therefore, we can use hardware that is specially taylored to linear operations: so-called *vector processors*. They are efficient for normal linear computations, and it turns out that they are very helpful for interval linear computations as well (see, e.g., [89]).

Vector processors are usually designed with operations on real numbers in mind. This design may be the best in terms of the speedup of linear operations with real numbers, but not in terms of operations of intervals. So, it is desirable to design a vector processor that will be specially suited for linear interval computations.

We have already mentioned that the typical linear operation $f(x_1, \ldots, x_n)$ consists of computing the value $c_1x_1 + \cdots + c_nx_n$. In other words, a typical linear operation is a *scalar* (dot) product $c \cdot x$, where $c = (c_1, \ldots, c_n)$ and $x = (x_1, \ldots, x_n)$. An ideal processor for computing an interval scalar product is described in [6] and [51].

Linear operations are important not only because we often need to estimate the value $f(X_1, \ldots, X_n)$ for a linear function f, but also because linear operations are part of other (more complicated) algorithms. These algorithms include different operations with vectors and matrices (e.g., solving a system of linear equations). [37] and [38] described how to design

a vector processor that would compute the matrix and vector operations fast and with correct interval estimates.

7.3 AI applications

One of the main sources of parallelism in AI applications is that in these applications, a knowledge base can be often divided into several weakly related parts. Therefore, we can assign to each part a processor that would handle this part of the database. The key, therefore, is the inter-processor connection scheme.

If two parts have something in common, so that from time to time, the corresponding processors need to exchange information, then we better make a direct link between them, so that the communication will take only one step. On the other hand, if two parts have practically nothing in common, and therefore hardly even communicate with each other, then it makes no sense to add a link (because an extra link has to be maintained, and this "eats up" additional computer time). So, the graph of connections between the processors must be ideally the same as the graph of connections between the parts. This specialized (domain-dependent) architecture is described, e.g., in [54] and [99].

7.4 Monte-Carlo methods

A specific feature of Monte-Carlo methods is that they practically do not require any communication at all: each of the processors computes f independently, and we only need communication at two moments of time:

- in the beginning, when we pass the data to all the processors, and
- at the end, when we collect the results y^k from all the processors, and process them to compute y.

Since there is no necessity for complicated protocols (that use processor time), so the ideal architecture for Monte-Carlo problems would require no protocols at all: we just pass the data, and then collect it. Such an architecture is described in [46–48].

If we must use a non-ideal architecture, we must design the routing algorithms that will be the best for Monte-Carlo applications. Such routing is described in [101] and [102].

7.5 Optimization problems

In addition to transputers [9], the main choice here is between central (master-slave) and truly distributed computing. Each of this methods has its advantages and disadvantages (a comparison is made in [32]).

- In *master-slave* configurations, the central processor is in control all the time.
  - The *advantage* is that the central processor has all the information about all the processors, and can thus make truly optimal decisions on the job distribution.
  - The *disadvantage* is that the only way for two neigboring processors to communicate with each other is to ask the permission of the central "bureaucrat".
- In *truly distributing* computing, procesors can freely communicate and share resources.
  - The main *advantage* is that this configuration enables the neighboring processors to share the workload with practically no time wasted on delayed communications with the center.
  - The main *disadvantage* is that in such a system, there is no one to see the big picture, and to suggest non-trivial optimal global job distribution.

An ideal possibility would be to *combine* the advantages of both approaches (this avoiding the drawbacks of both anarchy and autocracy), but how to do this is an open problem.

7.6 Neural networks

The very idea of a neural network, in which only two basic operations are performed, prompts the necessity for a special hardware. Several hardware

implementations of neural networks are known. A hardware idea that is taylored to interval neural computations is described in [27].

8 Software

Since parallellism is so important for interval computations, it would be nice to have a programming language for parallel interval computations. There exist several interval extensions of well-known programming languages (PASCAL–XSC, C–XSC, etc). A parallel extension of PASCAL–XSC is described in [14].

For further ideas, see [117]. For a more general parallel language, see [10, 11].

Acknowledgments

This work was partially supported by NSF grant No. CDA–9015006. The authors are greatly thankful to many researchers who helped with this special issue. Especially, we want to thank:

- the anonymous referees;
- Prof. Dr. J. Wolff von Gudenberg who organized a special session "Languages for parallel scientific computaion" at the CSAM'93 conference in St.Petersburg;
- Slava Nesterov for his encouragement;
- Alexander Yakovlev for his valuable comments, and
- Eldar Musaev for the complicated task of supervising this issue.

References

[1] Alefeld, G. and Herzberger, J. *On the convergence speed of some algorithms for the simultaneous approximation of polynomial roots.* SIAM Journal of Numerical Analysis **11** (1974), pp. 237–243.

[2] Alefeld, G. and Herzberger, J. *Über Simultanverfahren zur Bestimmung Reeler Polynomwurzeln.* Z. Angew. Math. Mech. **54** (1974), pp. 413–420.

[3] Babichev, A. B., Kadyrova, O. B., Kashevarova, T. P., Leshchenko, A.S., and Semenov, A. L. *UNICALC, a novel approach to solving systems of algebraic equations.* Interval Computations (2) (1993), pp. 29–47.

[4] Beeck, H. *Parallel algorithms for linear equations with not sharply defined data.* In: Feilmeier, M. (ed.) "Parallel Computers — Parallel Mathematics. Proceedings of the IMACS (AICA)–GI Symposium, March 14–16, 1977, Technical University of Munich", North Holland, Amsterdam, 1977, pp. 257–261.

[5] Bernat, A., Cortes, L., Kreinovich, V., and Villaverde, K. *Intelligent parallel simulation — a key to intractable problems of information processing.* In: "Proceedings of the 23-rd Annual Pittsburgh Conference on Modelling and Simulation, Pittsburgh, PA, 1992, Part 2", pp. 959–969.

[6] Bohlender, G. and Grüner, K. *Realization of an optimal computer arithmetic.* In: Kulisch, U. and Miranker, W. L. (eds) "A New Approach to Scientific Computation", Academic Press, Orlando, FL, 1983, pp. 247–268.

[7] Braes, D. and Hadeler, K. P. *Simultaneous inclusion of the zeros of a polynomial.* Numer. Math. **21** (1973), pp. 161–165.

[8] Caprani, O., Godthaab, B., and Madsen, K. *Use of a real-valued local minimum in parallel interval global optimization.* Interval Computations (2) (1993), pp. 71–82.

[9] Caprani, O. and Madsen, K. *Performance of an Occam/transputer implementation of interval arithmetic.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 12.

[10] Cooke, D. *A high level language to deal with multisets: discrete analog of intervals.* In: "Abstracts for a Workshop on Interval Methods in

Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 10.

[11] Cooke, D. *The term semantics of a high level language with applications to interval mathematics.* Interval Computations (1995) (to appear).

[12] Cortes, L. *How to design an expert system that for a given query Q, computes the interval of possible values of probability p(Q) that Q is true.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 11.

[13] Davidenkoff, A. *Parallel programming in PASCAL–XSC on a transputer system.* In: "Proceedings of the International Conference on Interval and Stochastic Methods in Science and Engineering INTERVAL'92", **2** 1992, Moscow, pp. 17–18.

[14] Davidenkoff, A. *Parallel programming in PASCAL–XSC.* Interval Computations (to appear).

[15] Eriksson, J. and Lidstrom, P. *A parallel interval method implementation for global optimization using dynamic load balancing.* Interval Computations (1995) (to appear).

[16] Fomin, Yu. I. and Cherkasov, A. A. *Parallel methods for solving the Cauchy problem for ordinary differential equations in interval implementation.* In: "Program Systems of Mathematical Physics. Proceedings of the 8th USSR National Workshop", Novosibirsk, 1984, pp. 62–69 (in Russian).

[17] Fomin, Yu. I. and Kodachigova, L. K. *A sweep method for linear algebraic system with strongly sparse tri-diagonal matrix.* VINITI, Moscow, Taganrog, 1988, Publ. No. 3692–B88 (in Russian).

[18] Fomin, Yu. I. and Kodachigova, L. K. *An interval sweep method for an interval linear algebraic system with strongly sparse tri-diagonal matrix.* VINITI, Moscow, Taganrog, 1988, Publ. No. 3694–B88 (in Russian).

[19] Gaganov, A. A. *Computational complexity of the range of the polynomial in several variables.* Cybernetics (1985), pp. 418–421.

[20] Gargantini, I. *Parallel algorithms for the determination of polynomial zeros.* In: Thomas R. S. D. and Williams H. C. (eds) "Proceedings of the 3rd Manitoba Conference on Numerical Mathematics, Winnipeg 1973, Part VIII", Utilitas Matematic Publ., 1974, pp. 195–211.

[21] Gargantini, I. *Parallel square root iterations.* In: Nickel K. (ed.) "Interval Mathematics", Lecture Notes in Computer Science **29**, Springer-Verlag, 1975, pp. 196–204.

[22] Gargantini, I. *Comparing parallel Newton's methods with parallel Laguerre's method.* Comput. Math. Appl. **2** (1976), pp. 201–206.

[23] Gargantini, I. *Parallel Laguerre iterations: the complex case.* Numer. Math. **26** (1976), pp. 317–323.

[24] Gargantini, I. *The numerical stability of simultaneous iterations via square-rooting.* Comput. Math. Appl. **5** (1979), pp. 25–31.

[25] Gargantini, I. *Parallel square-root iterations for multiple roots.* Comput. Math. Appl. **6** (1980), pp. 279–288.

[26] Garey, M. and Johnson, D. *Computers and intractability: a guide to the theory of NP-completeness.* Freeman, San Fransisco, 1979.

[27] Gulati, S., Gemoets, L., and Villaverde, K. *Error estimates for the results of intelligent data processing, especially neural networks.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 13.

[28] Hansen, E. R. *A generalized interval arithmetic.* In: K. Nickel (ed.) "Interval mathematics", Lecture Notes in Computer Science **29**, Springer-Verlag, 1975, pp. 7–18.

[29] Haque, A. L. M. S. *Parallel Laguerre's method for multiple zeros: implementation and test of the algorithm.* Master Thesis, Department of Computer Science, Faculty of Graduate Studies, University of Western Ontario, London, Canada, 1980.

[30] Henrici, P. *Uniformly convergent algorithms for the simultaneous determination of all zeros of a polynomial.* In: Ortega, J. W. and Rheinboldt, W. C. (eds) "Studies in Numerical Analysis 2, Numerical Solutions of Nonlinear Problems", SIAM, Philadelphia, 1970, pp. 1–8.

[31] Henr
the s
jon,
ment

[32] Henr
for p
Conf
neeri

[33] Herz
mit
(197

[34] Herz
zero
"Par
IMA
Marc

[35] Horn
netw
pp. 3

[36] JáJá
Read

[37] Kirch
"Pro
Italy.

[38] Kirch
Moo
Meth
1988

[39] Koda
algeb
Mosc

[40] Koda
paral

[31] Henrici, P. and Gargantini, I. *Uniformly convergent algorithms for the simultaneous approximation of all zeros of a polynomial.* In: Dejon, B. and Henrici, P. (eds) "Constructive Aspects of the Fundamental Theorem of Algebra", Wiley, London, 1969, pp. 77–113.

[32] Henriksen, T. and Madsen, K. *Combined real and interval methods for parallel global optimization.* In: "Proceedings of the International Conference on Interval and Stochastic Methods in Science and Engineering INTERVAL'92", **2** 1992, Moscow, pp. 30–32.

[33] Herzberger, J. *Über ein Verfahren zur Bestimmung Reeler Nullstellen mit Anwendung auf Parallelrechnung.* Elektron. Rechenanlangen **14** (1972), pp. 250–254.

[34] Herzberger, J. *Some multipoint-iteration methods for bracketing a zero with application to parallel computation.* In: Feilmeier, M. (ed.) "Parallel Computers — Parallel Mathematics, Proceedings of the IMACS (AICA)-GI Symposium, Technical University of Munich, March 14–16, 1977", North Holland, Amsterdam, 1977, pp. 231–234.

[35] Hornik, K., Stinchcombe, M., and White, H. *Multilayer feedforward networks are universal approximators.* Neural Networks **2** (1989), pp. 359–366.

[36] JáJá, J. *An introduction to parallel algorithms.* Addison-Wesley, Reading, MA, 1992.

[37] Kirchner, R. and Kulisch, U. *Arithmetic for vector processors.* In: "Proceedings of the 8th Symposium on Computer Arithmetic, Como, Italy, May 1987", IEEE Computer Society, 1987.

[38] Kirchner, R. and Kulisch, U. *Arithmetic for vector processors.* In: Moore, R. E. (ed.) "Reliability in Computing. The Role of Interval Methods in Scientific Computing", Academic Press, Boston, N.Y., 1988, pp. 3–41.

[39] Kodachigova, L. K. and Fomin, Yu. I. *A solution estimate for linear algebraic interval system by parallel interval sweep method.* VINITI, Moscow, Taganrog, 1988, Publ. No. 3341–B88 (in Russian).

[40] Kodachigova, L. K. and Fomin, Yu. I. *On stability of some methods parallelizing the sweep and the estimation of the solution width for*

a tri-diagonal interval linear algebraic system. In: Zyuzin, V. S., Ermakov, O. B., and Zakharov, A. V. (eds) "Proceedings of the Conference on Interval Mathematics, Saratov, May 23–25, 1989", pp. 22–25 (in Russian).

[41] Kohout, L. J. and Stabile, I. *Interval-valued inference and information retrieval in medical knowledge-based system CLINAID.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 16.

[42] Kohout, L. J. and Stabile, I. *Interval-valued inference in medical knowledge-based system CLINAID.* Interval Computations (3) (1993), pp. 88–115.

[43] Kohout, L., Stabile, I., Kalantar, H., San-Andres, M. F., and Anderson, J. *Parallel interval-based reasoning in medical knowledge-based system CLINAID.* Interval Computations (1995) (to appear).

[44] Koparkar, P. A. and Mudur, S. P. *A new class of algorithms for the processing of parametric curves.* Computer-Aided Design **15** (3) (1983), pp. 41–45.

[45] Koparkar, P. A. and Mudur, S. P. *Subdivision techniques for processing geometric objects.* In: Earnshaw, R. A. (ed.) "Fundamental Algorithms for Computer Graphics. Proceedings of the NATO Advanced Study Institute. Ilkley, Yorkshire, England, March 30 — April 12, 1985", Springer-Verlag, Berlin, Heidelberg, 1985, pp. 751–801.

[46] Kreinovich, V., Bernat, A., Villa, E., and Mariscal, Y. *Parallel computers estimate errors caused by imprecise data.* In: "Proceedings of the Fourth ISMM (International Society on Mini and Micro Computers) International Conference on Parallel and Distributed Computing and Systems, Washington, 1991" **1**, pp. 386–390.

[47] Kreinovich, V., Bernat, A., Villa, E., and Mariscal, Y. *Parallel computers estimate errors caused by imprecise data.* Interval Computations (2) (1991), pp. 31–46.

[48] Kreinovich, V., Bernat, A., Villa, E., and Mariscal, Y. *Parallel computers estimate errors caused by imprecise data.* In: "Technical Papers of the the Society of Mexican American Engineers and Scien-

tists 1992 National Symposium, San Antonio, Texas, April 1992", pp. 192–199.

[49] Kreinovich, V., Lakeyev, A. V., and Noskov, S. I. *Optimal solution of interval linear systems is intractable (NP-hard).* Interval Computations (1) (1993), pp. 6–14.

[50] Kreinovich, V. and Pavlovich, M. I. *Error estimate of the result of indirect measurements by using a calculational experiment.* Measurement Techniques **28** (3) (1985), pp. 201–205.

[51] Kulisch, U. and Bohlender, G. *Features of a hardware implementation of an optimal arithmetic.* In: Kulisch, U. and Miranker, W. L. (eds) "A New Approach to Scientific Computation", Academic Press, Orlando, FL, 1983, pp. 269–290.

[52] Kurkova, V. *Kolmogorov's theorem is relevant.* Neural Computation **3** (1991), pp. 617–622.

[53] Kurkova, V. *Kolmogorov's theorem and multilayer neural networks.* Neural Networks **5** (1992), pp. 501–506.

[54] Lassez, J.-L., Hyunh, T., and McAloon, K. *Simplification and elimination of redundant linear arithmetic constraints.* In: Lusk, E. L. and Overbeek R. A. (eds) "Logic Programming. Proceedings of the North American Conference", **1**, MIT Press, Cambridge, MA 1989, pp. 37–51.

[55] Layne, D. *Adaptive predictive control using neural networks and interval optimization.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", pp. 55–56.

[56] Leclerc, A. *Parallel interval global optimization in C++.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 57.

[57] Leclerc, A. *Parallel interval global optimization and its implementation in C++.* Interval Computations (3) (1993), pp. 148–163.

[58] Liu, J. W. S., Lin, K.-J., Shih, W.-K., and Yu, A. C.-S. *Algorithms for scheduling imprecise computations.* Computer (May 1991), pp. 58–68.

[59] Lovrenich, R. *Zone logic for manufacturing automation: intervals instead of optimization, goals instead of algorithms.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 22.

[60] Lyager, E. *Finding local extremal points using parallel interval methods.* Interval Computations (this issue) 1994.

[61] Madsen, K. and Toft, O. *A parallel method for linear interval equations.* Interval Computations (this issue) 1994.

[62] Menshikov, G. G. *Interval co-integration of differential equations connected by a substitution of the variable.* In: "Proceedings of the International Conference on Interval and Stochastic Methods in Science and Engineering INTERVAL'92", **2** (1992), Moscow, pp. 75–77.

[63] Mines, R., Nakamura, M., and Kreinovich, V. *Constructive proof of Kolmogorov's theorem, neural networks and intervals.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 24–25.

[64] Moore, R. E. *Interval analysis.* Prentice Hall, Englewood Cliffs, NJ, 1966.

[65] Moore, R. E. *Simple simultaneous super- and subfunctions.* In: Garloff, G. et al (eds) "Collection of Scientific Papers Honoring Prof. Dr. K. Nickel on Occasion of His 60th Birthday. Part I", Inst. f. Angew. Math., Universität Freiburg I. Br., 1984, pp. 259–274.

[66] Morlock, M. *Über das Newton-Simultanferahnen und Seine Anwendung auf die Abgebrochene Exponentialreihe.* Diploma, Institute for Informatics, University of Karlsruhe, 1969.

[67] Mudur, S. P. and Koparkar, P. A. *Interval methods for processing geometric objects.* IEEE Transactions on Computer Graphics and Applications **4** (2) (1984), pp. 7–17.

[68] Musa
ings c
1990"

[69] Musa
lems
Confe
sian).

[70] Musa
tional
Engin

[71] Musa
concu
60.

[72] Musa
repre
meric
Appli
1993"

[73] Naka
for K
works

[74] Narin
proce
(5) (1

[75] Narin
know
val M
Marc

[76] Newn
tion
on N
emati
Marc

[68] Musaev, E. A. *Wave computations in interval analysis.* In: "Proceedings of the Seminar on Interval Mathematics, Saratov, May 29–31, 1990", pp. 95–100 (in Russian).

[69] Musaev, E. A. *Hierarchical wave computations.* In: "Urgent Problems of Appied Mathematics. Proceedings of the USSR National Conference, Saratov, May 20–24, 1991, Part 1", pp. 110–112 (in Russian).

[70] Musaev, E. A. *Wave computations.* In: "Proceedings of the International Conference on Interval and Stochastic Methods in Science and Engineering INTERVAL'92", **2** (1992), Moscow, pp. 79–81.

[71] Musaev, E. A. *Wave computations. A technique for optimal quasi-concurrent self-validation.* Interval Computations (1) (1992), pp. 53–60.

[72] Musaev, E. A. *An approach to reliable computations with minimal representation.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", pp. 67–68.

[73] Nakamura, M., Mines, R., and Kreinovich V. *Guaranteed intervals for Kolmogorov's theorem (and their possible relation to neural networks).* Interval Computations (3) 1993, pp. 183–199.

[74] Narin'yani, A. S. *Subdefiniteness in the knowledge representation and processing system.* Izvestiya Acad. nauk SSSR, Tekhn. Kibernetika (5) (1986), pp. 3–28 (in Russian).

[75] Narin'yani, A. S. *NE-factors: different pragmatics of an interval in knowledge representation.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 26.

[76] Newman, F. and Cline, H. *A neural network for optimizing radiation therapy dosage.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 73.

[77] Park, C. Y. and Shaw, A. C. *Experiments with a program timing tool based on source-level timing schema.* Computer (May) 1991, pp. 48–57.

[78] Petkovic, M. S. *On the generalization of some algorithms for the simultaneous approximation of polynomial roots.* In: Nickel, K. (ed.) "Interval Mathematics", Academic Press, N.Y., 1980, pp. 461–471.

[79] Petkovic, M. S. *A family of simultaneous methods for the determination of polynomial complex zeros.* Internat. J. Comput. Math. **2** (1982), p. 285–296.

[80] Petkovic, M. S. *Generalized roor iterations for the simultaneous determination of multiple complex zeros.* Z. Angew. Math. Mech. **62** (1982), pp. 627–630.

[81] Petkovic, M. S. *On an iterative method for simultaneous inclusion of polynomial complex zeros.* J. Comput. Appl. Math. **8** (1982), pp. 51–56.

[82] Petkovic, M. S., Milovanovic, G. V., and Stefanovic, L. V. *On the convergence order of accelerated simultaneous method for polynomial complex zeros.* Z. Angew. Math. Mech. **66** (1986), pp. T428–T429.

[83] Petkovic, M. S. *On the simultaneous method of the second order for finding polynomial complex zeros in circular arithmetic.* Freiburger Intervall-Berichte (3) (1985), pp. 63–95.

[84] Petric, J., Jovanovic, M., and Stamatovic, S. *Algorithm for simultaneous determination of all roots of algebraic polynomial equations.* Mat. Vesnik **9** (1972), pp. 325–332.

[85] Plum, M. *Enclosures for solutions of parameter-dependent nonlinear elliptic boundary value problems: theory and implementation on a parallel computer.* Interval Computations (this issue), 1994.

[86] Rabinovich, S. *Measurement errors: theory and practice.* American Institute of Physics, N.Y., 1993.

[87] Schendel, U. *Einfuehrung in die Parallel Numerik. R.* Oldenbourg, Verlag, Muenchen-Vien, 1981.

[88] Sche
(Par

[89] Schm
"Abs
with
Softw

[90] Schn
val N
ical p
feren
Math
— M

[91] Senio
ings
leliza
(in R

[92] Shiri
"Abs
with
Softw

[93] Shvet
incon
Inter
ence
203 (

[94] Simci
Inter

[95] Sirisa
rame
Meth
Marc

[96] Sirisa
sensi
tions

[88] Scheu, G. *Über eine Wahl des Parameters beim Parellelverfahren (Parallel-Chord-Method).* Computing **20** (1978), pp. 17–26.

[89] Schmidt, L. *Vector processor support for semimorphic arithmetic.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 92.

[90] Schnepper, C. A. and Stadtherr, M. A. *Application of a parallel interval Newton/generalized bisection algorithm to equation-based chemical process flowsheeting.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 93.

[91] Senio, P. S. *Design of the interval Runge-like method.* In: "Proceedings of the 5th USSR National Conference and Workshop on Parallelization of Information Processing, Lvov, 1985, Part 4", pp. 50–51 (in Russian).

[92] Shiriaev, D. *Fast automatic differentiation for vector processors.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 98.

[93] Shvetsov, I. E. and Telerman, V. V. *Intervals and multi-intervals in incompletely defined computational models.* In: "Proceedings of the International Conference on Interval and Stochastic Methods in Science and Engineering INTERVAL'92", **1** (1992), Moscow, pp. 201–203 (in Russian; English abstract Vol. 2, p. 100).

[94] Simcik, L. and Linz, P. *Boundary-based interval Newton's method.* Interval Computations (4) (1993), pp. 89–99.

[95] Sirisaengtaksin, O. *Neural networks that are not sensitive to the parameters of neurons.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 30.

[96] Sirisaengtaksin, O. and Kreinovich, V. *Neural networks that are not sensitive to the imprecision of hardware neurons.* Interval Computations (4) (1993), pp. 100–113.

[97] Stefanovic, L. V. and Petkovic, M. S. *On the simultaneous improving k inclusive discs for polynomial complex zeros.* Freiburger Intervall-Berichte (7) (1982), pp. 1–13.

[98] Stefanovic, L. V. *Some modified methods for the simultaneous determination of polynomial zeros.* Dissertation, Faculty of Electronic Engineering, Nis, 1986 (in Serbo-Croatian).

[99] Swain, M. J. and Cooper, P. R. *Parallel software for constraint propagation.* In: "Proceedings AAAI–88 Seventh National Conference on Artificial Intelligence, Morgan Kaufmann" **2** (1988), pp. 682–686.

[100] Ullrich, C. and Reith, R. *A reliable linear algebra library for transputer networks.* Interval Computations (1995) (to appear).

[101] Villa, E. and Bernat, A. *Estimating errors of indirect measurements on real and realistic parallel machines.* In: "Abstracts for a Workshop on Interval Methods in Artificial Intelligence, Lafayette, LA, February 25 — March 1, 1993", p. 34.

[102] Villa, E., Bernat, A., and Kreinovich, V. *Estimating errors of indirect measurements on realistic parallel machines: routings on 2–D and 3–D meshes that are nearly optimal.* University of Texas at El Paso, Computer Science Department, Technical Report No. UTEP–CS–93–14a, 1993.

[103] Villaverde, K. *How to locate maxima and mimima of a function in parallel from approximate measurement results.* In: Kreinovich, V., Traylor, B., and Watson, R. (eds) "Abstracts of the First UTEP Computer Science Department Students Conference, El Paso, TX, 1991", pp. 43–44.

[104] Wang, D. and Wu, Y. *A parallel circular algorithm for the simultaneous determination of all zeros of a complex polynomial.* Freiburger Intervall-Berichte (8) (1984), pp. 57–76.

[105] Wang, D. *A parallel circular algorithm for the simultaneous determination of all zeros of a complex polynomial.* Journal of Engineering Mathematics (Xi'an) (1985), pp. 22–31 (in Chinese).

[106] Wang, X. and Zheng, S. *The quasi-Newton method in parallel circular iteration.* J. Comput. Math. **2** (1984), pp. 305–309.

[107] Wang
for fi
verge

[108] Wang
for fi
gence

[109] Wang
arith
Chin

[110] Wang
itera

[111] Wolfl
Inter

[112] Xing
arith
natio
Verif
Febr

[113] Yako
Kibe
Russ

[114] Yako
(inte
teria
Bran
pp. 3

[115] Yako
"Urg
Nati
(in F

[116] Yako
ges.
Stoc
(199

[107] Wang, X. and Zheng, S. *A family of parallel and interval iterations for finding all roots of a polynomial simultaneously with rapid convergence.* J. Comput. Math. **2** (1984), pp. 70–76.

[108] Wang, X. and Zheng, S. *A family of parallel and interval iterations for finding simultaneously all roots of a polynomial with rapid convergence. II.* Math. Numerica Sinica **4** (1985), pp. 433–444 (in Chinese).

[109] Wang, X. and Zheng, S. *Parallel Halley iteration method with circular arithmetic for finding all zeros of a polynomial.* Numer. Math. J. Chinese Univ. **4** (1985), pp. 308–314 (in Chinese).

[110] Wang, X. and Zheng, S. *Bell's disk polynomials and parallel disk iteration.* Freiburger Intervall-Berichte (2) (1986), pp. 37–65.

[111] Wolff von Gudenberg, J. *Parallel accurate linear algebra subroutines.* Interval Computations (1995) (to appear).

[112] Xing, Zh. and Shang, W. *Interval test: an application of interval arithmetic in data dependency analysis.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", p. 111.

[113] Yakovlev, A. G. *Machine arithmetic of multi-intervals.* Voprosy Kibernetiki (Problems of Cybernetics) **125** (1987), pp. 66–81 (in Russian).

[114] Yakovlev, A. G. *On some possibilities in organization of localizing (interval) computations on electronic computers.* Inf.-operat. material (interval analysis), preprint 16, Computer Center, Siberian Branch of the USSR Academy of Sciences, Krasnoyarsk (1990), pp. 33–38 (in Russian).

[115] Yakovlev, A. G. *Specific parallelism of localizing computations.* In: "Urgent Problems of Appied Mathematics. Proceedings of the USSR National Conference, Saratov, May 20–24, 1991, Part 1", pp. 151–158 (in Russian).

[116] Yakovlev, A. G. *Possibilities for further development of SC-languages.* In: "Proceedings of the International Conference on Interval and Stochastic Methods in Science and Engineering INTERVAL'92", **2** (1992), Moscow, pp. 134–136.

[117] Yakovlev, A. G. *Possibilities for further development of SC-languages.* In: "Abstracts for an International Conference on Numerical Analysis with Automatic Result Verification: Mathematics, Application and Software, Lafayette, LA, February 25 — March 1, 1993", pp. 112–113.

[118] Yakovlev, A. G. *Multiaspectness and localization.* Interval Computations (4) (1993).

Received: September 20, 1993
Revised version: August 29, 1994

Department of Computer Science
University of Texas at El Paso,
El Paso, TX 79968, USA
E-mail: vladik@cs.utep.edu
abernat@cs.utep.edu