Some new Acceleration Mechanisms in Verified Global Optimization

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

am Fachbereich Mathematik und Naturwissenschaften
Bergischen Universität Wuppertal

genehmigte

Dissertation

von

Jean Honoré Tapamo Kahou

Tag der mündlichen Prüfung:	24. November 2005
Referent:	Prof. Dr. A. Frommer
Korreferent:	Prof. Dr. B. Lang

Contents

List of Tables

List of Figures

Introduction

Optimization is ubiquitous in our daily live. From the way we organize our office to obtain more place to the angle the wing of a plane should have to obtain more strength, we always explicitly or implicitly solve optimization problems. The optimization problem is always addressed by scientific computing and applied mathematician researchers due to the huge demand coming from fields such as engineering or finance. In science, engineering and economics , decision problems are frequently modelled as optimizing the value of a (primary) objective (criterion, performance, loss etc.) function, under stated feasibility constraints to be met by all 'acceptable' decisions.
On the strict mathematical point of view, with regard to the type of problem (function), one can distinguish two types of optimization, namely *linear optimization* and *nonlinear optimization*. One talks about linear optimization when the function is linear in its variables, otherwise one talk about nonlinear optimization. While the linear optimization field is now a well searched field, with a rich literature and has many domains of application, the nonlinear optimization field can be considered partially searched and very difficult with regard to the huge CPU and memory requirement. We deal in this work with nonlinear optimization.
For many problems, according to a restricted search domain, there are solutions that do (that are satisfiable) there are called local optima. The best solution among all these solutions is the global optimum and one has the *global optimization* problem. Therefore, in contrast to *local optimization*, global optimization is concerned with finding the best optimum among all local optima.
Two approaches exist to solve the nonlinear global optimization problem. The first is the stochastic approach and the other is the deterministic approach. In general, starting from some approximate trial points, stochastic methods proceed by iteration. They sample the objective function at a finite number of points until some criterion is satisfied. Although widely applied, these methods lack robustness and are inherently unsuitable for 'verified complete search'. Indeed, the global optimum may escape detection when using traditional techniques due to a deep valley for example. The deterministic approach is based on the branch and bound principle and uses interval analysis. With interval analysis one is able to have a guaranteed enclosure of the result. Therefore, the use of interval analysis will serve two purposes, firstly the purpose of global convergence and secondly the purpose of auto-validation.
Verified global optimization requires a lot of computations and memory so that without appropriate acceleration mechanisms it may be considered untractable. The main subject of this thesis is to find some acceleration mechanisms to

speedup the convergence of the interval global optimization algorithm.
Here is the organization of this thesis. In the first chapter we give an overview of interval analysis - the basic tool for verified global optimization - and its properties.
In the second chapter we do an in-depth analysis of the interval global optimization problem. The algorithm we use belongs to the branch and bound category. By branch and bound we mean that the initial search domain is subdivided into smaller parts and these parts are searched for the global optimum. But these parts are not uniformly searched, instead some parts are preferred. We present in this chapter two mechanisms to speed up the convergence of the interval global optimization algorithm. The first mechanism is concerned with an appropriate management of subproblems arising during the search process. As a matter of fact, for many difficult problems a great part of the whole computation time is spent on the handling of a list, which for example can have millions of elements. This is definitely an issue one has to deal with when solving global optimization problems using interval analysis. Experimental results for the mechanism we propose to cope with this issue show sometimes a dramatic decrease of the computation time.
The second mechanism is concerned with a so called one dimensional Newton iteration. This test is based on the fact that for many problems, a fairly good approximation for global minimum is known relatively early in the search process. It also uses the fact that - due to the smoothness of the function under consideration - the value of the gradient is available via the monotonicity test or the use of centered forms. The aim of this test is then to apply one iteration of the Newton method to shrink or discard the box under consideration. This test relies on the knowledge of the approximation of the global minimum, the better the approximation, the more powerful the test. Experimental results show interesting improvements when applying this test along with others.
The third and last chapter is concerned with the investigation of a new parallelization strategy. Whereas for many linear algebra problems, the amount of work at each node can be estimated at the beginning, it is very difficult (almost impossible) to do the same for branch-and-bound algorithms. This is due to their irregular and unpredictable computational behavior. It is then clear that, static load balancing, very often efficient for many linear algebra problems, will become inefficient for branch-and-bounds algorithms. In this chapter, we first make a review of existing methods dealing with the parallelization of the interval global optimization algorithm. We then present a new technique (*distributed management*) to evenly load jobs among processors during the computation

process. The strength of this new approach relies on the distribution of the task of the root processor among other processors. With this new approach the root (master) will hardly become a bottleneck.
Algorithms presented in this thesis are implemented in C++ using the interval library CXSC see [45]. Parallel algorithms were implemented using the Message Passing Interface (MPI) library. The environment for the parallel implementation is the Alpha Linux Cluster Engine at Wuppertal University see Section(3.1.2). Appendix A gives the description of the problems considered in this thesis.

Notations

We describe in the following main typographic conventions and symbols used in the thesis.

Sets

R	:	set of machine numbers
$\mathbb{R}$	:	set of real numbers
$\mathbb{IR}$	:	set of all interval real numbers
$\mathbb{IR}^n$	:	set of all interval vectors (box)
$\mathbb{IR}^{n\times m}$	:	set of all interval matrices
$\mathcal{L}$	:	list, stack, queue, heap
$\mathbb{I}$(D)	:	elements of $\mathbb{IR}^n$ contained in D

Intervals

$[x] = [\underline{x}, \overline{x},]$	:	an interval
$[x] = ([x]_i)_{i=1,\ldots,n} = ([x]_1, \ldots, [x]_n)^T$	:	an interval vector or **box**
$[A] = ([a]_{ij})\ _{j=1,\ldots,m}^{i=1,\ldots,n}$	:	interval matrix
$\inf([x])$ or $\underline{x}$	:	lower bound of $[x]$
$\sup([x])$ or $\overline{x}$	:	upper bound of x
$m([x])$	:	midpoint or center of $[x]$
$w([x])$	:	width of $[x]$
$mig([x])$	:	mignitude of $[x]$
$\|[x]\|$	:	absolute value or magnitude of $[x]$
$q([x],\ [y])$	:	Hausdorff distance between $[x]$ and $[y]$

Although an interval vector is denoted in the same manner as a scalar interval, there should be no confusion. When we will be using box, we will state it explicitly.
Operations inf, sup, w, m are defined componentwise on interval vectors and matrices. For example $m([x]) = (m([x_i]), \ldots, m([x_n]))$.

Functions

f	:	the objective function.
f'	:	the derivative or gradient of f
f''	:	the second derivative or Hessian of f
$\frac{\partial f(x_1,\ldots,x_n)}{\partial x_i}$	:	partial derivative of f
f^*	:	the global minimum
$\tilde{f}$	:	an approximation for the global minimum
$\square f([x])$	:	the range of f over the box $[x]$
$F([x])$	:	interval extension of f

Other symbols

$\lozenge$	:	interval rounding
$\square$	:	rounding to the nearest element of a floating point screen
$\triangledown$	:	rounding toward $-\infty$ or downwardly directed
$\triangle$	:	rounding toward $+\infty$ or upwardly directed
$\overset{\circ}{[x]}$	:	interior of $[x]$

Acknowledgments

Here, I would like first of all to thank Professor Andreas Frommer for his substantial help through this work. A special thank to Professor Bruno Lang for its collaboration and advice. Many thanks to Mrs. Shultz and my colleagues Holger Arndt, Thomas Beelitz, Klarsten Blankenangel, Elton Bojaxhiu, Stefan Borovac, Peter Feurtstein, Sigrid Fischer, Matthias Hüsken, Stefanie Krisky, Peter Langer, Katrin Schäfer and Paul Willems for their collaboration. I would also like to thank all the members of my family and my friends for their invaluable support .

Chapter 1

Overview of Interval Analysis

Interval arithmetic allows one to bound the range of a function over a domain, it is therefore a tool of choice for global optimization. This ability that interval analysis has to give all information about a function over a domain is a key point in verified global optimization. Interval analysis is so important for global optimization that it deserves a special section in this work. However, we will not go deeply in details in its description, we would rather present some important and key features that it has. For more details about interval analysis one can consult the literature [18, 16, 45, 46, 34].
A real compact interval, we will simply call it interval, is an non-empty closed and bounded subset of the real numbers $\mathbb{R}$

$$[x] = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\},$$

where $\underline{x}$ and $\overline{x}$ denote the *lower* and the *upper* bounds of the interval $[x]$, respectively. The interval $[x]$ consists of the set of points between and including its endpoints. If $\underline{x} = \overline{x}$ then we have a point interval also called *degenerated* or *thin* interval; otherwise the interval is called a *thick* interval. The set of intervals is denoted $\mathbb{IR}$.
Let $D \subseteq \mathbb{R}$, we define

$$\mathbb{I}(\mathrm{D}) = \{[x] : \ \ [x] \in \mathbb{IR} \ \text{ and } \ [x] \subseteq D\}$$

So $\mathbb{I}(\mathrm{D})$ is the set of all intervals included in D. In the next section we extend operations available for real numbers to intervals.

1.1 Interval arithmetic

Let $* \in \{+, -, \cdot, \div\}$ be a binary operation on the set of real numbers $\mathbb{R}$ and let $[x]$ and $[y] \in \mathbb{IR}$. We set

$$[x] * [y] = \{z = x * y \mid x \in [x], \, y \in [y]\} \tag{1.1}$$

which defines these binary operations in $\mathbb{IR}$. This definition produces the following rules for generating endpoints for $[x] * [y]$ from $[x]$ and $[y]$. (We first suppose that $0 \notin [y]$ for the case of the division $\div$. If $0 \in [y]$ then we obtained the extended interval arithmetic to be presented later).

$$[x] \ * \ [y] = \left\{ \begin{array}{lcl} [x] + [y] & = & [\underline{x} + \underline{y}, \, \overline{x} + \overline{y}] \\ [x] - [y] & = & [\underline{x} - \overline{y}, \, \overline{x} - \underline{y}] \\ [x] \cdot [y] & = & [\min\{\underline{x}\underline{y}, \, \underline{x}\overline{y}, \, \overline{x}\underline{y}, \, \overline{x}\overline{y}\}, \, \max\{\underline{x}\underline{y}, \, \underline{x}\overline{y}, \, \overline{x}\underline{y}, \, \overline{x}\overline{y}\}] \\ [x] \div [y] & = & [x] \cdot [1/\overline{y}, \, 1/\underline{y}] \end{array} \right. \tag{1.2}$$

In the above definitions we excluded the division by an interval containing 0. For the case where $0 \in [y]$, we obtain the following expressions for $[x] \div [y]$, allowing the result to be the union of two possibly infinite intervals.

$$[x] \div [y] = \begin{cases} \left[\overline{x}/\underline{y},\ \infty\right) & \text{if } \overline{y} \leq 0 \text{ and } \overline{y} = 0 \\ (-\infty, \overline{x}/\overline{y}] \cup \left[\overline{x}/\underline{y},\ \infty\right) & \text{if } \overline{y} \leq 0 \text{ and } \underline{x} < 0 < \overline{y} \\ (-\infty,\ \underline{x}/\overline{y}] & \text{if } \overline{x} \leq 0 \text{ and } \underline{y} = 0 \\ (-\infty,\ \infty) & \text{if } \underline{x} < 0 < \overline{y} \\ \left(-\infty,\ \underline{x}/\underline{y}\right] & \text{if } \underline{x} \geqslant 0 \text{ and } \overline{y} = 0 \\ \left(-\infty, \underline{x}/\underline{y}\right] \cup [\underline{x}/\overline{y},\ \infty) & \text{if } \underline{x} \geqslant 0 \text{ and } \underline{y} < 0 < \overline{y} \\ [\underline{x}/\overline{y},\ \infty) & \text{if } \underline{x} \geqslant 0 \text{ and } \underline{y} = 0 \end{cases} \tag{1.3}$$

The definition of these operations on the set $\mathbb{IR}$ allows one to manipulate intervals as we usually manipulate reals, defining an arithmetic on $\mathbb{IR}$. But the properties of these operations on $\mathbb{IR}$ are not the same as their counterparts in $\mathbb{R}$.

1.1.1 The dependency problem

Suppose one wants to compute endpoints of the interval $[x] - [x]$ with the definition presented above. One obtains

$$[x] - [x] = [\underline{x} - \overline{x},\ \overline{x} - \underline{x}]$$

and not 0. This is due to the fact that, with the definition of the subtraction, the set computed is $\{x - y,\ x \in [x],\ y \in [x]\}$ instead of $\{x - x,\ x \in [x]\}$. This occurs in general when an expression contains more than one occurrence of a variable. Those occurrences are treated in fact as if there were different variables from the same interval, resulting in a widening of the result. There exist special procedures to reduce the effect of this *dependency problem*, see [16] and references therein.
We have defined binary operations on $\mathbb{IR}$, in the next paragraph we do the same for unary operations.

1.1.2 Unary operations in $\mathbb{IR}$

Let $r(x)$ be a unary operation defined on $D \subseteq \mathbb{R}$, let $[x] \subseteq D$. Then

$$r([x]) = [\min_{x \in [x]} r(x),\ \max_{x \in [x]} r(x)]$$

defines its unary counterpart in $\mathbb{IR}$.
Examples of such operations are cos, sin, x^k, $k \in \mathbb{R}$, log etc. For example

$$[x]^n = \begin{cases} [1,\,1] & \text{if } n = 0 \\ [\underline{x}^n,\,\overline{x}^n] & \text{if } (\underline{x} \geq 0) \text{ or } (\underline{x} \leq 0 \leq \overline{x} \text{ and } n \text{ odd}) \\ [\overline{x}^n,\,\underline{x}^n] & \text{if } \overline{x} \leq 0 \text{ and } n \text{ even} \\ [0,\,\max\{\underline{x}^n,\,\overline{x}^n\}] & \text{if } \underline{x} \leq 0 \leq \overline{x} \text{ and } n \text{ even} \end{cases}$$

Unary operations are interval valued functions depending on one interval variable. The generalization to functions of many variables will be discussed in section 1.4.

1.1.3 Real-valued functions of an interval

There are a lot of real-valued functions of an interval. Here we list those we are going to use throughout this thesis.

The **midpoint** or the **center** of an interval $[x]$ is

$$m([x]) = \frac{\overline{x} + \underline{x}}{2} .$$

The **width** of an interval $[x]$ is

$$w([x]) = \overline{x} - \underline{x} .$$

The **absolute value** or the **magnitude** of an interval $[x]$ is

$$| [x] | = \max\{| \underline{x} |, | \overline{x} |\} .$$

The **mignitude** of an interval $[x]$ is

$$mig([x]) = \begin{cases} \underline{x} & \text{if } \underline{x} > 0 \\ -\overline{x} & \text{if } \overline{x} < 0 \\ 0 & \text{otherwise} \end{cases}$$

The *mignitude* is the minimum value of $| x |$ for all $x \in [x]$.

The **Hausdorff distance** between two intervals is

$$q([x],\,[y]) = \max\{| \underline{x} - \underline{y} |, | \overline{x} - \overline{y} |\} .$$

For more real-valued functions of intervals and the relations among them see [18]. In the next section we present some properties of interval operations.

1.1.4 Properties of the operations in $\mathbb{IR}$

Let $[x]$, $[y]$, $[z]$ be members of $\mathbb{IR}$, it follows that

$$\begin{array}{rcl} [x] + [y] & = & [y] + [x] \\ [x] \cdot [y] & = & [y] \cdot [x] \end{array} \qquad (Commutativity).$$

$$\begin{array}{rcl} ([x] + [y]) + [z] & = & [x] + ([y] + [z]) \\ ([x] \cdot [y]) \cdot [z] & = & [x] \cdot ([y] \cdot [z]) \end{array} \quad (Associativity).$$

$[a] = [0, 0]$ and $[b] = [1, 1]$ are the unique neutral elements with respect to addition and multiplication. $\mathbb{IR}$ has no zero divisors.
If $\underline{x} \neq \overline{x}$ then $[x]$ has no inverse with respect to $+$ and $\cdot$. But $0 \in [x] - [x]$ and $1 \in [x] \div [x]$.
This is one of the differences between $\mathbb{R}$ and $\mathbb{IR}$ and the origin of the *dependency problem*. One consequence of this difference is the absence of the distributivity. In the case of interval analysis one only has a subdistributive property stated by the following expression.

$$[x] \cdot ([y] + [z]) \subseteq [x] \cdot [y] + [x] \cdot [z] \qquad (Subdistributivity).$$

Proofs of these rules can be found in [18].

The *intersection*, *union* and set relations for intervals are defined as for sets. The interior of an interval $[x]$ is denoted by $\overset{\circ}{[x]}$ and is defined as the interval without its bounds. Therefore, $a \in \overset{\circ}{[x]} \Leftrightarrow \underline{x} < a < \overline{x}$.

In the next section we examine the multidimensional case.

1.2 Interval vectors and matrices

An interval vector is a vector whose elements are intervals. An interval matrix is a matrix whose elements are intervals. The set of all n-dimensional interval vectors is denoted by $\mathbb{IR}^n$. In the same manner, $\mathbb{IR}^{n \times m}$ denotes the set of all real interval matrices. We use the notations

$$[x] = ([x]_i)_{i=1,\ldots,n} = ([x]_1, \ldots, [x]_n)^T, \quad \text{for } [x] \in \mathbb{IR}^n$$

and

$$[A] = ([a]_{ij})_{\substack{i=1,\ldots,n \\ j=1,\ldots,m}} = \begin{pmatrix} [a]_{11} & \cdots & [a]_{1m} \\ \vdots & & \vdots \\ [a]_{n1} & \cdots & [a]_{nm} \end{pmatrix}.$$

A real interval vector may be interpreted as the set of points in n-dimensional space bounded by a parallelepiped with sides parallel to the coordinate axes. For this reason we will sometimes, to be short, call an interval vector a *box*. Many operations and functions defined in $\mathbb{IR}$ are defined in $\mathbb{IR}^n$ componentwise. For example the *midpoint* of a box $[x]$ is $m([x]) := (m([x]_i))$. An exception is the *width* of a *box* which is the width of the edge with the maximal width, that is, for $[x] \in \mathbb{IR}^n$

$$w([x]) = \max_{1 \le i \le n} w([x]_i).$$

1.3 Implementation of interval arithmetic

Interval arithmetic as presented above requires exact arithmetic to compute the endpoints of the resulting intervals. But if we want to implement interval arithmetic on a computer we have to face the fact that computers have only a finite set of numbers that are often represented in a semilogarithmic manner as fixed length floating point numbers

$$x = m \cdot b^e.$$

Here m is the mantissa, b the base, and e the exponent. The numbers are normally represented internally with base $b = 2$ and a normalized mantissa, that is $\frac{1}{2} \le \mid m \mid < 1$. The integer exponent e is bounded by $e_{min} \le e \le e_{max}$. The set of machine numbers of the above type, the floating point *screen*, is denoted by R. We now denote the set of floating-point intervals over R by

$$IR = \{[x] \in \mathbb{IR}, \mid \underline{x}, \overline{x} \in R\}.$$

This definition means that a machine interval $[x] \in IR$ denotes the continuum of numbers lying between its bounds. Probably one of the most important characteristic of floating-point interval arithmetic is that, when computing with floating machine numbers we obtain results holding not only for every *floating-point* number but also for every real number within that range.
To achieve this, we need a rounding

$$\Diamond : \mathbb{IR} \to IR$$

which maps an interval to a machine interval. This *interval rounding* should satisfy the following conditions.

$$\begin{array}{rl} \Diamond [x] = [x] & \text{for all } [x] \in IR \\ [x] \subseteq [y] \Rightarrow \Diamond [x] \subseteq \Diamond [y] & \text{for all } [x], [y] \in \mathbb{IR} \\ \Diamond(-[x]) = -\Diamond [x] & \text{for } [x] \in \mathbb{IR} \end{array}$$

The first condition guarantees that elements of the screen are not changed by a rounding. The second means that a rounding is *monotone*, and the third means that the rounding is *antisymmetric*. Moreover the following condition must be satisfied

$$[x] \subseteq \Diamond([x]).$$

This is the most crucial requirement for machine interval arithmetic, as we will explain later in Section 1.5. One distinguishes the following roundings for real numbers

$\Box$: Rounding *to the nearest* element of R

∇ : Rounding *toward* $-\infty$ or *downwardly* directed

Δ : Rounding *toward* $+\infty$ or *upwardly* directed

The interval rounding $\Diamond$ can then be achieved by rounding the upper toward $+\infty$ and the lower bound toward $-\infty$
An elementary floating-point interval operation is defined by

$$[x] \diamondsuit [y] \;=\; \Diamond([x] * [y]) \quad \text{for all } [x],\ [y] \in IR,$$

where $*$ is an interval arithmetic operation, $* \in \{+, -, ., / \div\}$
In a simple manner we extend unary operations r to unary machine operations by setting

$$\Diamond[x] \;=\; \Diamond(r[x]).$$

As a consequence of this definition, machine interval arithmetic guarantees that the computed result of an expression will contain the range of this expression interpreted as a function on its input interval. This sets the ground why machine interval arithmetic can be used to obtain verified computational results, i.e results which have the same rigor as a mathematical proof, see Section 1.5.
There are many libraries that implement a machine interval arithmetic with the rounding requirements. One can cite **C-XSC** (C++ Class Library for eXtended Scientific Computing), **filib** see [45], INTLIB see chapter 2 in [26] and IntLab (Interval Laboratory) see [48]. We use the C-XSC library in this thesis. We give in the next paragraph an overview of its major functionalities, for an extensive presentation see [45, 11].

1.3.1 The C-XSC library

C-XSC is a tool for solving scientific problems with automatic verification of the result. It is available for personal computers, workstations and mainframes due to its implementation as a C++ class library.
C-XSC supports additional features for safe programming such as index range checking for vectors and matrices. It also supports checking for numerical errors such as overflow, underflow, loss of accuracy, illegal arguments, etc.
All arithmetic operators provided by C-XSC deliver results of maximum accuracy. The mathematical standard functions deliver results of high accuracy. Moreover, C-XSC provides the possibility to evaluate dot product expressions with maximum accuracy. The evaluation of such expressions is the fundamental tool for solving sensitive numerical problems.

Now that we have defined binary and unary operations on $\mathbb{IR}$ and we know how to implement them, we are now able to extend the definition of a real function f to intervals.

1.4 Functions of intervals

An *interval function* is an interval-valued function of one or more interval arguments. A *natural interval extension* of a real-valued function f is a function F obtained by replacing, in the expression of f, all real variables and constants by intervals. It follows that the following condition will be satisfied

$$F(x_1, \ldots, x_n) \;=\; f(x_1, \ldots, x_n) \tag{1.4}$$

for all x_i $(i \;= 1, \ldots, n)$. That is, if the arguments of F are degenerate intervals, then $F(x_1, \ldots, x_n)$ is a degenerate interval equal to $f(x_1, \ldots, x_n)$, provided exact arithmetic is used.

Definition 1 *Let $f\colon D \subseteq \mathbb{R}^n \to \mathbb{R}$ be a function. Then $F\colon \mathbb{I}(\mathrm{D}) \to \mathbb{IR}$ is called* ***inclusion function*** *of f if*

$$f(x_1, \ldots, x_n) \;\in\; F([x_1], \ldots, [x_n])$$

whenever $x_i \in [x_i],\ i = 1, \ldots, n$.

We define the range of a real function f over $[y] \in \mathbb{IR}^n$ as

$$\Box f([x]) = \{f(x) :\ x \in [y]\}.$$

It follows from Definition (1) that

$$\Box f([y]) \subseteq F([y]) \tag{1.5}$$

will be satisfied if F is an inclusion function of f. Because of the properties of interval arithmetic presented earlier, natural interval extensions are special inclusion functions.
Property (1.5) is the key of almost all interval arithmetic applications, it is sometimes called the *fundamental property of interval arithmetic*, see [18].

Definition 2 *An inclusion function F is said to be inclusion isotonic or inclusion monotonic over $[x]$ if for $[y]$, $[z] \in \mathbb{I}([x])$, $[y] \subseteq [z]$ implies $F([y]) \subseteq F([z])$.*

Definition 3 *Let $[x] \subseteq \mathbb{R}^n$ be a box and $F : \mathbb{I}([x]) \to \mathbb{IR}$ an interval valued function then F has the zero convergence property if $w(F([y])) \to 0$ as $w([y]) \to 0$ for $[y] \in \mathbb{I}([x])$.*

Unless otherwise stated, we shall assume that any interval function used in the sequel is inclusion isotonic. Since a real function may be expressed in many ways, it follows that a given real function has many natural interval extensions.

Example 1 *Let f be the real function defined by $f(x) = x^2 - 2x$, $x \in \mathbb{R}$. The following interval functions are inclusion functions of f.*

- $F_1([x]) = [x]^2 - 2[x]$,
- $F_2([x]) = [x]([x] - 2)$,
- $F_3([x]) = ([x] - 1)^2 - 1$.

With $[x] = [-2, 3]$, one obtains

- $F_1([x]) = [-6, 13]$,
- $F_2([x]) = [-12, 8]$,
- $F_3([x]) = [-1, 8] = \square f([x])$.

A measure of the quality of an inclusion function F of f is the *excess-width*,

$$w(F([y])) - w(\square f([y]))$$

for $[y] \in \mathbb{IR}^n$ introduced by Moore, see [16].
An inclusion function F of f is said to have *(convergence) order* $\alpha > 0$ if

$$w(F([y])) - w(\square f([y])) = \mathcal{O}(w([y])^\alpha),$$

meaning that there exists a constant $c > 0$ such that

$$w(F([y])) - w(\square f([y])) < c \cdot w([y])^{\alpha} \quad \text{for } [y] \in \mathbb{IR}^n.$$

In order to obtain fast computational methods in optimization it is important to choose inclusion functions of an order α as high as possible when $w([y])$ becomes small. A detailed investigation of the order of inclusion functions is given in [16, 46]. A similar looking concept, which is however different of the order, is the idea of a Lipschitz condition.
Let $D \subseteq \mathbb{IR}^n$ and $F : \mathbb{I}(\mathrm{D}) \to \mathbb{IR}$. Then F is called *Lipschitz* if there exists a real number K (*Lipschitz constant*) such that

$$w(F([y])) \leq K \cdot w([y]) \quad \text{for } [y] \in \mathbb{IR}^n.$$

The Lipschitz property delivers us a frequently used criterion for the meanvalue form which is a special inclusion function of convergence order 2, see [46]. We present this special inclusion function in the next section.

1.4.1 Centered forms, Meanvalue forms, Taylor forms

When evaluating an inclusion function one has in general two choices. One can choose natural interval extensions as presented above or *centered forms*. Centered forms are inclusion function with special features that were introduced by Moore [16]. The most important centered forms are the *meanvalue form* and the *Taylor form*.

Meanvalue forms

Let $D \subseteq \mathbb{IR}^m$ be open and

$$f : D \to \mathbb{R}, \quad D \subseteq \mathbb{R}^m$$

be differentiable and let

$$F' : \mathbb{I}(\mathrm{D}) \to \mathbb{IR}^m$$

be an inclusion function for the gradient f' of f. Then

$$T_1 : \mathbb{I}(\mathrm{D}) \to \mathbb{IR}$$

defined by

$$T_1(c, [y]) = f(c) + ([y] - c)^T F'([y]) \quad \text{for } [y] \in \mathbb{I}(\mathrm{D}), \tag{1.6}$$

where $c \in [y]$ is called the *meanvalue form function* or shorter: *meanvalue form.* The point c is frequently taken equal to $m([y])$, the midpoint of $[y]$. However, more sophisticated choices are possible by which one can minimize the upper bound of the interval $T_1(c, [y])$ or maximize its lower bound, see Baumann [46]. Frequently, F' will be computed as a natural interval extension of f' via *automatic differentiation arithmetic*, see [11, 43]

Theorem 1 *If F' is Lipschitz, then the meanvalue form T_1 is of convergence order 2.*

A proof can be found in many books dealing with interval arithmetic, e.g. [18].

Example 2 *The following example from [46] gives us an idea about the qualitative difference between orders of convergence of the meanvalue form and the natural interval extension of function.*

Let $f(x) = x - x^2$ be defined on $D = \{x : x \geq 1\} \subseteq \mathbb{R}$. An inclusion function for $f'(x) = 1 - 2x$ is

$$F'([y]) = 1 - 2[y] \text{ for } [y] \in \mathbb{I}(\mathrm{D}).$$

With $c = m([y])$, the meanvalue form of f is

$$T_1(c, [y]) = (c - c^2) + ([y] - c)(1 - 2[y]) \text{ for } [y] \in \mathbb{I}(\mathrm{D}).$$

The natural interval extension of $f(x)$ is

$$F([y]) = [y] - [y]^2 \text{ for } [y] \in \mathbb{I}(\mathrm{D}).$$

The range of f over D is

$$\Box f([y]) = [x^2 - x,\, y^2 - y] \text{ for } [y] = [x,\, y] \in \mathbb{I}(\mathrm{D}).$$

Let us now calculate the widths of the inclusion functions and the width of the range. We have

$$\begin{aligned} w(\Box f([y])) &= y^2 - y - (x^2 - x) = y^2 - x^2 - (y - x) \\ &= w([y])(y + x - 1). \end{aligned}$$

Using the fact that $w([a][b]) = 2\overline{a}\max\{|\ \underline{b}\ |, |\ \overline{b}\ |\}$ whenever $0 \in [a]$, and the fact that

$$\max\{|\ 1 - 2x\ |, |\ 1 - 2y\ |\} = 2y - 1, \quad (1 \leq x \leq y)$$

one obtains

$$\begin{aligned} w(T_1(c,\,[y])) &= w[([y]-c)(1-2[y])] \\ &= w([y]-c)(2y-1) \\ &= w([y])(2y-1). \end{aligned}$$

And the width of the natural interval extension is

$$\begin{aligned} w(F([y])) &= (y-x)+(y^2-x^2) \\ &= w([y])(y+x+1). \end{aligned}$$

A short calculation shows that

$$w(T_1(c,\,[y])) \leq w(F([y])), \text{ iff } w([y]) \leq 2.$$

This means that the meanvalue form is superior for small intervals. In this example we have

$$w(T_1(c,\,[y])) - w(\square f([y])) = w([y])^2 = O(w([y])^2)$$

$$w(F([y])) - w(\square f([y])) = 2 \cdot w([y]) = O(w([y])).$$

This is consistent with the fact that the meanvalue form is of convergence order 2, but the interval extension is only of order 1. Yet an other example.

Example 3 *Let $f(x) = x^2 - x$. Suppose we evaluate this function over the interval $[x] = [-1,\,3]$. With the natural interval extension one obtains*

$$F([x]) = [x]^2 - [x] = [-3,\,10]$$

and with Taylor form one obtains

$$T_1(c,\,[x]) = T_1(1,\,[x]) = F(m([x])) + ([x] - m([x]))F'([x]) = [-10,\,10].$$

But if one evaluates the same function over the interval $[x] = [0.3,\,0.6]$, one obtains

$$F([x]) = [x]^2 - [x] = [-0.51,\,0.06], \quad w(F([x])) = 0.57$$

and

$$T_1(c,\,[x]) = [-0.29,\,0.18], \quad w(T_1(c,\,[x])) = 0.47$$

These examples show that is not always wise to use the meanvalue form. In general it is not advantageous to use the meanvalue form for large width intervals. In our numerical experiments we use meanvalue form (or in general centered forms) when the width of the box under consideration is less $1/2$. We now discuss another centered form.

Taylor forms

Let $D \subseteq \mathbb{IR}^m$ be open and

$$f : D \to \mathbb{R},\ D \subseteq \mathbb{R}^m$$

be twice differentiable, and let

$$F'' : \mathbb{I}(\mathrm{D}) \to \mathbb{IR}^{m \times m}$$

be an inclusion function for the Hessian matrix f'' of f. Then

$$T_2 : \mathbb{I}(\mathrm{D}) \to \mathbb{IR}$$

defined by

$$T_2(c,\, [y]) = f(c) + ([y] - c)^T f'(c) + \frac{1}{2}([y] - c)^T F''([y])([y] - c) \tag{1.7}$$

for $[y] \in \mathbb{IR}^m$ where $c \in [y]$, is called *Taylor form function* (or simply *Taylor form*) for f of second order. The mean value form (1.6) may be regarded as a Taylor form of first order. We have the following theorem.

Theorem 2 *If f is twice differentiable, and $|F''([y])| \leq d$ for all $[y] \in \mathbb{I}(\mathrm{D})$, then the Taylor form function T_2, is of convergence order two.*

For a proof see [18]. If m is large then the computation of F'' becomes expensive, and its explicit computation should be avoided. Some techniques to deal with this issue have been investigated, see [46].

Example 4 *We take the same function as in the previous example, that is $f(x) = x^2 - x$. We take $[x] = [-a,\ a]$ and $c = 0$ we have*

$$\Box f([-a,\ a]) = [a^2 - a,\ a^2 + a],$$

$$T_1(c,\, [x]) = [x] \cdot (2 \cdot [x] - 1),$$

and

$$T_1(0, [-a,\ a]) = [-a,\ a] \cdot ([2a - 1,\ -2a - 1]).$$

$$T_2(c,\, [x]) = [x] \cdot (2 \cdot [x] - 1) + [x]^2,$$

and

$$T_2(0, [-a,\ a]) = [-a^2 - a,\ a^2 + a].$$

Now we can compare the widths:

$$w(\square f([-a,\ a])) = 2a,$$

$$w(T_1(0,\ [-a,\ a])) = 2a \cdot \max\{|2a-1|,\ |2a+1|\} = 2a \cdot (2a+1),$$

$$w(T_2(0,\ [-a,\ a])) = 2a \cdot (a+1).$$

yielding

$$w(T_2(0,\ [-a,\ a])) - w(\square f([-a,\ a])) = 2a^2 = \frac{1}{2}(w([a]))^2,$$

$$\begin{aligned} w(T_1(0,\ [-a,\ a])) - w(\square f([-a,\ a])) &= 2a \cdot \max\{|2a-1|,\ |2a+1|\} - 2a \\ &= 4a^2 = (w([a]))^2. \end{aligned}$$

We see that there is no improvement in the convergence order when passing from T_1 to T_2.

An other technique related to centered forms has been investigated in the literature [22], the so-called *mixed centered inclusion function*. We give here a brief description, for more details see [22].

Mixed centered inclusion functions

The main idea to obtain the mixed centered inclusion function is to apply (1.6) n times, considering each variable of the function in turn. We expose this technique, for the sake of simplicity, for the case $n = 3$.
Consider $f(x_1, x_2, x_3)$ as a function of x_3 only and take $m_3 = m([x_3])$; we obtain, applying (1.6)

$$f(x_1, x_2, x_3) \in f(x_1, x_2, m_3) + g_3(x_1, x_2, [x_3]) * ([x_3] - m_3)$$

where g_3 is the partial derivative of the function with respect to the third variable. Consider furthermore $f(x_1, x_2, m_3)$ as a function of x_2 only and take $m_2 = m([x_2])$; One obtains using (1.6)

$$f(x_1, x_2, m_3) \in f(x_1, m_2, m_3) + g_2(x_1, [x_2], m_3) * ([x_2] - m_2).$$

Finally, consider $f(x_1, m_2, m_3)$ as a function of x_1 and take $m_1 = m([x_1])$; then (1.6) yields

$$f(x_1, m_2, m_3) \in f(m_1, m_2, m_3) + g_1([x_1], m_2, m_3) * ([x_1] - m_1)$$

Combining these three equations one obtains

$$\begin{aligned} f(x_1,x_2,x_3) \quad \in \quad & f(m_1,m_2,m_3) + g_1([x_1],m_2,m_3) * ([x_1]-m_1) \\ + \quad & g_2(x_1,[x_2],m_3) * ([x_2]-m_2) \\ + \quad & g_3(x_1,x_2,[x_3]) * ([x_3]-m_3). \end{aligned}$$

It follows that

$$\begin{aligned} \Box f([x_1],[x_2],[x_3]) \quad \subset \quad & f(m_1,m_2,m_3) + g_1([x_1],m_2,m_3) * ([x_1]-m_1) \\ + \quad & g_2([x_1],[x_2],m_3) * ([x_2]-m_2) \\ + \quad & g_3([x_1],[x_2],[x_3]) * ([x_3]-m_3). \end{aligned}$$

This expression can be generalized for a function f of n variables. With $x = (x_1,\ldots,x_n)^T$ and $m = m([x])$, one gets

$$\Box f([x]) \subset f(m) + \sum_{i=1}^{n} [g_i]([x_1],\ldots,[x_i],m_{i+1},\ldots,m_n)\cdot([x_i]-m_i). \qquad (1.8)$$

The right hand side of (1.8) defines the *mixed centered inclusion function.*
Mixed centered inclusion functions can be viewed as a special case of *slope functions*, definition of which is given below.
A slope sf of a function f w.r.t a point m is defined as a function from $\mathbb{R}^n$ to $\mathbb{R}^n$ such that

$$f(x) - f(m) = sf(x,\, m)(x-m) \qquad \text{for all } x.$$

If SF is an inclusion function for sf, then

$$\Box f([x]) \subseteq f(m) + SF([x],\, m)\cdot([x]-m).$$

For more details see [41, 34]. One advantage with slopes is that there is no requirement concerning the smoothness of the function. Therefore one can implement derivatives free global optimization algorithms.

1.5 Principles of numerical verification

This is one of the virtues interval analysis has; namely the automatic verification of numerical results. The easiest way is probably to replace any real or complex operation by its interval equivalent and then perform the computations using interval arithmetics. The procedure leads to reliable, verified results. However, the width of the computed enclosures may be too wide to bee practically useful.

In general one therefor applies mechanisms, using interval arithmetic, to get a verified result from an already computed approximate solutions. To achieve this, many algorithms for numerical verification are based on the application of well known fixed-point theorems to intervals sets. The following theorem can be found in [11].

Theorem 3 (Brouwer's fixed-point theorem) *Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a continuous mapping and $X \subseteq \mathbb{R}^n$ a non-empty, closed, convex and bounded set. If $\square f(X) \subseteq X$, then f has at least one fixed-point x^* in X.*

A box $[x]$ in n-dimensional space, satisfies the conditions of Brouwer's fixed-point theorem. So, if an inclusion function F for f satisfied $F([x]) \subseteq [x]$ we have $\square f([x]) \subseteq [x]$, consequently f has a fixed point x^* in $[x]$. This theorem could be applied in our context to check the uniqueness of a local minimizer in a subbox $[y]$ of the starting box $[x]^0$. For more details see [11].

Chapter 2

The Global Optimization Problem

The aim of this chapter is to present and discuss in detail the different aspects of interval global optimization algorithms. In particular we will present new strategies that we have developed to speed-up the convergence of such algorithms.
We want to find the global minimum in a given area of $\mathbb{R}^n$ of a function f. Global minimum here is by opposition to local minimum. This means that we are looking for the smallest of all local minima if there exist several of them. The theory and analysis of global optimization algorithms can be considered to be relatively new in comparison to the local optimization theory for which there exists a rich reference in the literature; see the references in [51]. Since the global maximum of the function f is the global minimum of the function $-f$, global minimization is equivalent to global maximization, so that we restrict ourselves to global minimization here. We also want to find *all* points where the global minimum is reached. Global minimization or global optimization in the sequel, is not an easy task for methods which use information of the function at a finite number of points only, because narrow, deep valleys may escape detection. In contrast, the interval method presented here evaluates the function f on a continuum of points, including those points that are not finitely representable, so valleys, no matter how narrow, are never neglected.
Algorithms to solve the global optimization problem can be divided into two big groups, namely, *stochastic* and *deterministic* methods.

2.1 Stochastic methods

In general, starting from some approximate trial points, stochastic methods proceed by iteration. They sample the objective function at a finite number of points until some criterion is satisfied. Stochastic methods included *simulated annealing*, *evolutionary algorithms* and *clustering methods*. For more details about these methods one can see [19]. The advantage of these methods is that they have a reasonable complexity which is why they are sometimes preferred in practice. But since these methods sample the objective function at only a finite number of points, they cannot guarantee that the global minimum has been found.

2.2 Interval methods

Interval global optimization methods are the only ones which yield guaranteed information about the global minimum and the points where this would be achieved. These methods do so by producing an interval which is known to

contain the minimum value and a set of boxes which contains all possible minimizers. The aim of these methods is to discard parts of the search domain that cannot contain global minimizers.

2.3 Problem statement

Given is a function

$$f : \mathbb{I}(\mathrm{D}) \subset \mathbb{R}^n \to \mathbb{R}.$$

The aim is to find the minimum (provided it exists)

$$f^* = \min_{x \in D} f(x) \tag{2.1}$$

subject to

$$g_i(x) \leq 0, \quad i = 1, \ldots, k,$$

$$h_i(x) = 0, \quad i = k+1, \ldots, r,$$

where $g_i, h_i : D \to \mathbb{R}$,
and the set

$$S^* = \{x \in D : \quad f(x) = f^*\}$$

where this minimum is reached. The functions g_i, h_i are called constraints. If $r = 0$, then the problem is said to be *unconstrained*. In this work we always assume $r = 0$. If one prescribes an initial domain (box) where the minimum is to be found, then one has these types of constraints

$$a_i \leq x_i \text{ and } x_i \leq b_i, \quad i = 1, \ldots, n.$$

In this case, the problem is said to be with *bounds constraints*. This is exactly the situation which will be considered in this work. Usually, we will also assume that f is twice differentiable on D.

2.4 Interval global optimization methods

To solve the problem (2.1) the interval methods use the branch and bound principle. By branch and bound we mean here that the given problem is divided into several subproblems which themselves might be further subdivided recursively (branching). When working on the (most promising) subproblems, a criterion is dynamically updated which allows to discard some of the subproblems since one knows that they do not contain the solution to the original problem (bounding). For the box-constrained global optimization problem, subproblems are generated by subdividing the current box $[x] \in \mathbb{IR}^n$ into smaller subboxes $[x]^1, \ldots, [x]^l$

(starting with $[x] = [x]^0$) and by considering the global optimization problems on the smaller boxes. Using interval arithmetic, as described in the previous chapter, a lower bound for f over each such box $[y]$ is computed. At the same time, evaluating f at carefully chosen points, we know an upper bound $\tilde{f}$ for the global minimum of f. Bounding is now done by discarding the subproblems for those boxes $[y]$ where the lower bound for $\Box f([y]$ is large than $\tilde{f}$.
Proceeding in this manner, one generates a tree of subproblems in which the bounding principle prevents certain subproblems to be subdivided. Figure 2.1 illustrates this by showing the subdivision of the boxes which in this case are bisected along their largest side.

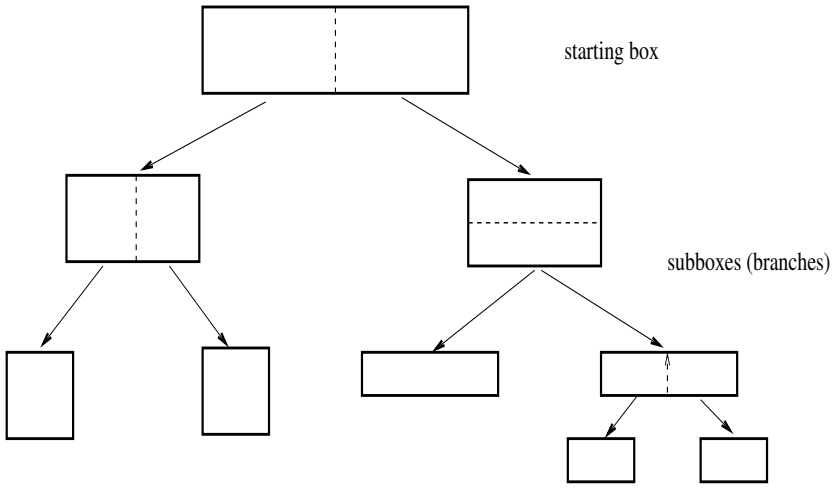

Fig. 2.1: Illustration of the branch and bounds principle

The algorithm we present next has been investigated, with some slight differences, by many authors, see Hansen [19], Moore-Skelboe [46] and Ichida-Fujii [46].
An algorithmic framework for interval methods for global (unconstrained) optimization consists of

- the basic steps
- the accelerating devices

The basic steps are responsible for getting the solution of the problem or, at least, an approximation. The aim of the accelerating devices is to obtain the solution as fast possible. Below are the steps of the algorithm. The algorithm begins with some initializations. The working list is set to the starting box

algorithm 1 Interval Branch & Bound Algorithm for Global Optimization

1: **Input:** $[\mathbf{x}]^{\mathbf{0}}$ starting box, ϵ tolerance for the stopping criterion,
2: F inclusion function for f
3: **Output:** $\tilde{f}$, approximation for f^* and $\mathcal{S}$, list of boxes covering S^*
4: $\tilde{f} = \overline{F}(m([x]^0))$
5: initialize work list $\mathcal{L} = ([x]^0, \underline{F}([x]^0))$, solution list $\mathcal{S} = \varnothing$
6: **while** $\mathcal{L}$ is not empty **do**
7: **choose** a pair $P = ([x], \underline{F}([x]))$ from $\mathcal{L}$
8: **if** stopping criterion holds **then**
9: insert P into $\mathcal{S}$ and **goto 6**
10: **end if**
11: $\tilde{f} = \min\{\tilde{f}, \overline{F}(mid([x]))\}$ {update the minimum}
12: split the box $[x]$ into $([x]_1, \ldots, [x]_n)$ { sub-dive [x]}
13: compute $F([x]_i)$ for $i = 2 \ldots n$ and store $([x]_i, \underline{F}([x]_i))$ in $\mathcal{L}$
14: apply acceleration devices on $[x]_1$
15: { monotonicity test, convexity test, Newton step, ... }
16: perform **cut_off** test on $\mathcal{L}$
17: insert what remains of $[x]_1$ into $\mathcal{L}$
18: **end while**

$[x]^0$ and the result list of boxes containing the global minimizers equals the empty set. The algorithm subdivides parts of the starting box recursively and updates the value of $\tilde{f}$, the upper bound of the global minimum. As we have said before, the algorithm consists of the basic part and the accelerating devices. The basic part is the part without the accelerating devices which include the monotonicity, convexity and the Newton test. These accelerating devices will be presented separately. But before doing so, we give some general results about the convergence, the complexity and the stopping criteria of Algorithm 1.
The cut_off test is a procedure that removes some elements of the working list for which one knows that they do not contain global minimizers. Details on this test will be given in 2.5.1.

2.4.1 Convergence, complexity, and stopping criteria

Stopping criteria

In our algorithm a box $[x]$ is inserted in the solution list S if it satisfies the following condition

$$w(F([x])) \leq \epsilon_F, \tag{2.2}$$

where ϵ_F is the tolerance on the function values. Some authors, [16], also require that

$$w([x]) \leq \epsilon_x, \tag{2.3}$$

where ϵ_x is the tolerance on the box. This second condition makes sense when one wants to have global minimizers within a certain accuracy. In general the first condition is sufficient to have a reasonable enclosures of both the minimum and the minimizers. The first condition (2.2) can be relaxed and is sometimes replaced by $\tilde{f} - \underline{F}([x]) < \epsilon_F$.

Proposition 1 *If Algorithm 1 terminates, then $\mathcal{S}^* \subseteq \mathcal{S}$, where $\mathcal{S}^*$ is the set of all global minimizers of f.*

Proof: For the purpose of this proof we regard the lists $\mathcal{L}$ and $\mathcal{S}$ as sets consisting of the union of their respective boxes, so that $\mathcal{S}^* \subseteq \mathcal{S}$ actually makes sense. We now prove that at any iteration of the algorithm we have $\mathcal{S}^* \subseteq \mathcal{L} \cup \mathcal{S}$. This is true before the first iteration, since $\mathcal{L} = [x]^0$ and $\mathcal{S} = \emptyset$. Assume that it is true for some iteration. When lines 8-10 are executed, the set $\mathcal{L} \cup \mathcal{S}$ does not change since we only shift boxes from $\mathcal{L}$ to $\mathcal{S}$. When $[x]$ is split in line 12, the boxes $[x]_2, \ldots, [x]_n$ are inserted into L, and $[x]_1$ may be modified by acceleration devices. Since the acceleration devices are all such that all global minimizers from $[x]_1$ will still be contained in what remains from $[x]_1$, we have that $\mathcal{S}^*$ is contained in the union of $\mathcal{L}$, $\mathcal{S}$ and what remains from $[x]_1$ after the acceleration devices. The cut_off test on line 16 can not remove a box containing a global minimizer. On the line 18, what remains from $[x]_1$ is inserted in $\mathcal{L}$. It follows that we still have $\mathcal{S}^* \subseteq \mathcal{L} \cup \mathcal{S}$. Upon termination of the *while* loop, we have $\mathcal{L} = \emptyset$, so that $\mathcal{S}^* \subseteq \mathcal{S}$. □

Proposition 2 *Assume that for Algorithm 1 we use the stopping criterion* (2.3) *with $\epsilon_x > 0$. Then Algorithm 1 terminates.*

Proof: Assume that the algorithm does not terminate. Let $[y]^k$ be the sequence of boxes with $[y]^k$ the box selected in the k-th iteration. Let $s([y]^k)$ be the number of subdivision steps performed, starting from $[x]^0$, to obtain $[y]^k$. Then $s([y]^k) \leq s* = m \cdot \lceil \log_n \frac{w([x]^0)}{\epsilon_x} \rceil$, because otherwise $w([y]^k) \leq \epsilon_x$ and that $([y]^k)$ would have been moved to S. Here n denotes the number of subdivision parts and m is the dimension. But there exist only $\sum_{k=0}^{s^*} n^k$ boxes $[y]$ with $s([y]) \leq s^*$, so that the boxes $[y]^k$ can not be all different. But this is a contradiction, since a box $[y]^k$ which has been selected is never inserted again into $\mathcal{L}$. □

Proposition 3 *Assume that F is zero convergent and the stopping criterion is taken as* (2.2)*, i.e.*

$$w(F([x])) < \epsilon_F,$$

then Algorithm 1 terminates.

Proof: Since F is zero-convergent, there exists $\epsilon_x \geq 0$ such that $w([x]) \leq \epsilon_x \Rightarrow w(F([x])) \leq \epsilon_F$ for all $[x] \subseteq [x]^0$. With this assertion the proof follows in a manner completely analogous to Proposition (2). □
Propositions 1 - 3 give the conditions under which Algorithm 1 is correct. Proposition 1 shows that no global minimizer is lost during the search process. Proposition 2 shows that condition (2.3) is enough to guarantee the termination. Proposition 3 relies on the fact that condition (2.3) implies condition (2.2) if F is zero convergent. Note that Proposition (2)- (3) are valid when one subdivides the box along the coordinate with maximal interval width. More details about subdivision strategies will be given in Section 2.4.6.

Complexity

The interval global optimization algorithm requires in general a lot of computational resources so that other methods are sometimes preferred in practice. But recall that it is the only method that can claim to solve the global optimization problem (that can guarantee that the global solution has been found). It is obvious that if all accelerating devices fail then the algorithm is exponential in the dimension of the starting box. The interval global optimization problem is even NP-hard since the basic problem of computing exact bounds for the range of a function is NP-hard, see [19] and references therein.

Convergence of the algorithm

To investigate the convergence of Algorithm 1 we suppose that the stopping criterion will never be fulfilled. In this case Algorithm 1 is equivalent to Algorithm 3 on page 111 in [46]. To have the same settlements for Algorithm 1 as those in [46], we denote by $([y]_n, \tilde{y}_n)$ the box with $\tilde{y}_n = \min\{\underline{F}([x])\}$, for $[x] \in \mathcal{L}_n$, where n is the iteration index. The working list at the iteration n is denoted by $\mathcal{L}_n$. We denote further by U_n the union of boxes in the list $\mathcal{L}_n$. We have the following theorems concerning the convergence of Algorithm 1.

Lemma 1 *Let $([y]_n)_{n=1}^{\infty}$ be a sequence generated by Algorithm 1, then*

$$w([y]_n) \to 0 \text{ as } n \to \infty.$$

The proof of this lemma is similar to the one of proposition 2 and can be found in [46], page 85.

Theorem 4 *If the inclusion function F in Algorithm 1 has the zero convergence property, then sequence $(F([y]_n))_{n=1}^{\infty}$ converges to f^*.*

For a proof, see [46], page 86.

Theorem 5 *If the inclusion function F in Algorithm 1 has the zero convergence property, then $U_n \supseteq S^*$ for all n and $U_n \to S^*$ as $n \to \infty$. The sequence (U_n) is nested and thus $S^* = \cap_{n=1}^{\infty} U_n$.*

For proof, see [46], page 113.

2.4.2 Selection strategies

The first important point of the algorithm presented above is the way one selects the next box to process. Many strategies have been investigated in the literature on how to chose the next box. Here we present these strategies and we also present a new strategy we developed. One distinguishes

- The *oldest-first* strategy: choose the oldest box, i.e. the work list $\mathcal{L}$ is handled as a FIFO (First In First Out) queue.
- The *best-first* strategy: choose the box $[x]$ with the smallest lower bound $\underline{F}([x])$, i.e. $\mathcal{L}$ is handled as priority queue.
- The *depth-first* strategy: choose one of the most recently created boxes, i.e. $\mathcal{L}$ is handled as a stack.
- The *reject-index* strategy: choose the box where a quantity to be defined is the largest. As with the *best-first* strategy, $\mathcal{L}$ is a priority queue in this case too.

Oldest-first strategy. In the *oldest-first* strategy, the next box to process is the box that has spent the most time in the working list. In this case, the algorithm implements $\mathcal{L}$ as a simple queue. The box at the head is the box to select and new boxes are inserted at the tail. The first advantage of this strategy is that all boxes are regularly subdivided. Therefore, one can expect the width of the function over the boxes to tend to zero rapidly when the number of iterations of the algorithm grows. The second advantage of this strategy is that the management of the list is very simple and also efficient. Any operation

on the list is done in constant time. This is definitely a very important practical point in interval global optimization algorithm. In fact as the size of the list grows, for some difficult test problems, where the work list is very large, the algorithm may spend most of the time handling the list, the (computation of functions or derivatives) becoming negligible. The disadvantage is that this strategy does not favor promising boxes, so that some boxes which by other strategies would have been discarded due to the *cut-off* test will remain in the work list and will further be processed.

Best-first strategy. Widely used, this strategy favors the box where the function has the smallest lower bound, i.e. the pair $([x], \underline{F}([x]))$ of the work list with the smallest second element is chosen. The idea is that the algorithm will then always be working on the most promising box. Therefore, a good approximation of the global minimum will be reached relatively early. Consequently the *cut-off* test and other tests based on the quality of $\tilde{f}$ will be more efficient. A proposition emphasizing the importance of this strategy is given below and can be found in [1].

Theorem 6 *Using Algorithm 1 with the* best-first *strategy, no pair* $([x], \underline{F}([x])$ *with* $\underline{F}([x]) > f^* + \epsilon_F$ *will be chosen for subdivision.*

Proof, see [2].
The drawback with this strategy is that the time to manage the list as a priority queue can become very noticeable for the algorithm on some problems. We will come back to this point later.

Depth-first strategy. In the *depth-first-strategy*, one manages boxes in a LIFO (Last In First Out) data structure. The algorithm thus implements a stack. To proceed, one always takes the first box from the stack and places new subboxes into the stack. In this way the box will be subdivided further and further until the termination criteria are fulfilled.
One of the advantages of this strategy is that it maintains a short list (stack) and that furthermore, the update operations on the list are in constant time.
With this strategy, boxes could be subdivided unnecessarily. This happens for example when one only inserts the part of the box which contains the global minimizer into the work list. The other part would be subdivided unnecessarily.

Reject-index based strategy. Recently introduced, ([9, 6]), this selection strategy is based on a quantity called the *reject index* and noted pf_k. This

quantity is defined as following

$$pf_k = \frac{f_k - \underline{F}([x])}{\overline{F}([x]) - \underline{F}([x])}$$

where f_k is an known approximation of the global minimum value at the iteration k which will be studied in more details later. With this strategy, the next box to process is the box where this quantity is maximal. The reject index estimates the relative position of the global minimum value of the objective function within the range given by the inclusion function. This quantity should then indicate whether the given interval $f([x])$ is likely to contain a minimizer. The motivation is mainly as follows ([6]): Traditionally, the *best-first* strategy is used, therefore the box $[x]$ with the smallest $\underline{F}([x])$ is considered as the best candidate to contain a global minimum. However, usually, the larger the box $[x]$, the larger the overestimation of $\square f([x])$ in $F([x])$. Therefore, in the *best-first* strategy, a box could be considered as a best candidate to contain a global minimizer just because it is larger than others. In order to compare subboxes with different size one normalizes the distance between $\tilde{f}$ and $\underline{F}([x])$. The idea behind is that one expects the overestimation to be symmetric, i.e. the overestimation above $f([x])$ is almost equal to the overestimation below $f([x])$, for small subboxes containing a global minimizer point. For more details and theoretical investigations see [9, 6].

2.4.3 New selection strategies

We have implemented the *reject-index* and the *best-first* strategies on many problems. The *reject-index* has a little advantage on the *best-first* on some problems. Looking closer on the tables below we see that the maximal length reached by the list when using the *reject-index* is almost always smaller than the maximal length reached by the list when using of the *best-first* strategy. It seems that the advantage the *reject-index* has is that the list is maintained small. We present below some new selection strategies which maintain small list while favoring promising boxes.

The relative reject-index

It is known that using the *oldest-first* strategy, the list remains in general smaller than using another strategy. However, the algorithm with the *oldest-first* strategy converges slowly, since it does not favor promising boxes. It would then be advantageous to think of a strategy that could do both, namely, favor the promising box and keep the list small.

As with the *reject-index* presented above, we now propose to chose the next box (pair) to process in Algorithm 1, the box for which the *relative reject-index*

$$rf_k = \frac{f_k - \underline{F}([x])}{(\overline{F}([x]) - \underline{F}([x])) * (w([x]))}$$

is maximal. One can see that rf_k is obtained by multiplying pf_k by $\frac{1}{w([x])}$. Therefore, rf_k is maximal when pf_k is maximal and $w([x])$ is minimal. In this way, this new selection strategy can be seen as a combination of the reject-index and the depth-first strategy, and would (hopefully) have the advantages of both. According to our experimental results, The version of Algorithm 1 implementing the relative reject-index is in general better the than the worst version of the algorithm using the *best-first* and *reject-index*, see the experimental results below. We will analyze the convergence of the algorithm with this new selection strategy later.

The hybrid selection strategy

Since with the best-first strategy one obtains a good approximation for the global minimum relatively early and, with the depth-first strategy the list is maintained small, it would be interesting to combine the two strategies. One could then begin with best-first and finish with depth-first. One switches from best-first to depth-first when a good approximation for the global minimum is reached. We call such strategy a *hybrid selection strategy*. Of cause, the question arises as to know when to switch. One issue is to check whether the value of $\tilde{f}$ does not change during a certain number of iterations, taking this as an indicator that $\tilde{f}$, obtained so far, is a good approximation for the global minimum. The problem with this approach is that it depends too much on the problem under consideration, as our numerical experiments showed. Another way out is to check whether the length of the working list decreases, because in this case it is likely also that a good approximation of the minimum is reached. Our tests indicate that this is often the case for difficult test problems. We therefore opted for this second choice, and the experimental results presented below show that using this strategy, one obtains improvements, we compare these strategies later. The next section deals with the convergence of the algorithm when the reject-index or the new selection strategies is used.

Convergence properties of the algorithm using the *reject-indices* and the hybrid selection strategy

Since both versions of the algorithm using the best-first and the oldest-first converge, it is clear, by the construction of the hybrid selection strategy, that

the resulting algorithm will converge.
Now we investigate the convergence conditions of the algorithm using either of the reject-index strategies. For this purpose we present Theorem 1 in [10] and show that it is applicable in the case of the relative reject-index too. In [10], the next box to process is the box $[y]$ for which the reject index

$$pf(f_k, [y]) = \frac{f_k - \underline{F}([y])}{\overline{F}([y]) - \underline{F}([y])},$$

is maximal. Herein, we have a choice for f_k which we assume to be between the best known lower bound and upper bound $\tilde{f}_k$ of f^*, i.e. we define

$$\underline{f}_k = \min\{\underline{F}([y]^l),\ l = 1, \ldots, |\mathcal{L}|\} \leq f_k < \tilde{f}_k = \overline{f}_k. \tag{2.4}$$

$|\mathcal{L}|$ denotes the current number of elements in the list $\mathcal{L}$.

Theorem 7 ([10]) *Assume the inclusion function of the objective function is isotone and it has the zero convergence property. Consider further Algorithm 1 above in which the next pair to process is the pair* $([y],\ \underline{F}([y]))$ *from the working list* $\mathcal{L}$ *which has the maximal value* $pf(f_k, [y])$. *The boxes thus selected will be called leading boxes. We assume that the stopping criterion is never fulfilled.*

1. *Each accumulation point of the sequence of leading boxes is a global minimizer of* f *if*

$$\underline{f}_k \leq f_k < \delta(\overline{f}_k - \underline{f}_k) + \underline{f}_k \tag{2.5}$$

   holds for each iteration number k, *where* $0 < \delta < 1$ *is fixed.*

2. *Condition* (2.5) *is sharp in the sense that* $\delta = 1$ *allows convergence of* $\tilde{f}$ *to a value larger than* f^*.

Proof: 1. Since f_k is not less than the minimal lower bound $\underline{f}_k$ of F, it follows that the maximal $pf(f_k, [y])$ values are always nonnegative. The numerator of pf is less than $\tilde{f} - \min\{\underline{F}([y]^l),\ l = 1, \ldots, |\mathcal{L}|\}$ since $f_k < \tilde{f}$. The sequence $\underline{f}_k$ is monotonously nondecreasing and $\overline{f}_k$ is monotonously non-increasing since F is isotone.
Now consider an arbitrary point $x' \in [x]$ in such a way that $f(x') > f^*$, and assume that there is a subsequence $\{[y]_{k_l}\}$ of leading boxes that converges to x'. We have that

$$f(x') = \lim_{l \to \infty} f([y]_{k_l}) = \lim_{l \to \infty} \underline{F}([y]_{k_l})$$

since f is continuous and F has the zero convergence property.
Further, since

$$\tilde{f}_k \leq \min_{[y]\in\mathcal{L}_k} \overline{F}([y])$$

because of the update of f_k, we have

$$\tilde{f}_{k_l} \leq \overline{F}([y]_{k_l}).$$

The sequence $\tilde{f}_{k_l}$ is monotonously decreasing and bounded below by f^*. Therefore,

$$\lim_{l\to\infty} \tilde{f}_{k_l}$$

exists and due once more to the zero-convergence property, one has

$$\lim_{l\to\infty} \tilde{f}_{k_l} \leq f(x').$$

With this and according to (2.5) one has

$$\begin{aligned} f_{k_l} - \underline{F}([y]_{k_l}) &= (f_{k_l} - \underline{f}_{k_l}) + (\underline{f}_{k_l} - \underline{F}([y]_{k_l})) \\ &< \delta(\tilde{f}_{k_l} - \underline{f}_{k_l}) + (\underline{f}_{k_l} - \underline{F}([y]_{k_l})) \\ &= \delta(\tilde{f}_{k_l} - \underline{F}([y]_{k_l})) + (1-\delta)(\underline{f}_{k_l} - \underline{F}([y]_{k_l})) \\ &= \delta(\tilde{f}_{k_l} - \overline{F}([y]_{k_l})) + \delta(\overline{F}([y]_{k_l}) - \underline{F}([y]_{k_l})) \\ &\quad + (1-\delta)(\underline{f}_{k_l} - \underline{F}([y]_{k_l})) \end{aligned}$$

Herein, we have

$$\tilde{f}_{k_l} - \overline{F}([y_{k_l}]) \leq 0 \text{ for all } l.$$

Due to the zero convergence property

$$\lim_{l\to\infty} (\overline{F}([y]_{k_l}) - \underline{F}([y]_{k_l})) = 0.$$

Since

$$\lim_{l\to\infty} \underline{f}_{k_l} \leq f^* \text{ and } \lim_{l\to\infty} \underline{F}([y]_{k_l}) = f(x'),$$

we know that for all $l > l_0$ one has

$$\underline{f}_{k_l} - \underline{F}([y]_{k_l}) \leq \frac{1}{2}(f^* - f(x')).$$

For l sufficiently large we have

$$\underline{f}_{k_l} - \underline{F}([y]_{k_l}) \leq 0.$$

Since $0 < \delta < 1$ this means that the respective pf values are negative from an index K. This contradicts the fact that the maximale pf values are always nonnegative
2. The second statement is a consequence of Theorem 3 and Corollary 2 in [8]. Since the correctness of this proof does not depend on the value of the denominator of pf, this remains true for the *relative reject-index* and we have the following theorem.

Theorem 8 *Assume the inclusion function of the objective function is isotone and it has the zero convergence property. Consider further Algorithm 1 above in which the next pair to process is the pair* $([y], \underline{F}([y]))$ *from the working list* $\mathcal{L}$ *which has the maximal value* $rf(f_k, [y])$.

1. *Each accumulation point of the sequence of leading boxes is a global minimizer of* f *if*
$$\underline{f}_k \leq f_k < \delta(\overline{f}_k - \underline{f}_k) + \underline{f}_k \tag{2.6}$$
holds for each iteration number k, *where* $0 < \delta < 1$ *is fixed.*

2. *Condition* (2.6) *is sharp in the sense that* $\delta = 1$ *allows convergence of* $\tilde{f}$ *to a value larger than* f^*.

2.4.4 Handling the list

Numerical experiments show that the length of the working list can have a big influence on the run-time. This is particularly the case when $\mathcal{L}$ is a priority queue. Note that, in this case, the *cut-off* is easily done, but each insert operation has a complexity of $\mathcal{O}(|\mathcal{L}|)$, where $|\mathcal{L}|$ is the length of the current list. We would like to minimize the influence of handling the list. Therefore, we have to implement a data structure that will allow us to do all operations in less than $\mathcal{O}(|\mathcal{L}|)$. The idea is to avoid all operations with linear cost on the list. Since at any moment a particular box is preferred, we have to deal with a priority queue. The efficient and straightforward data structure to think about is a heap. All the element produced by the algorithm are therefore stored in the heap. Consequently all operations are now in $\mathcal{O}(\log(|\mathcal{L}|))$. The only drawback with the heap is that performing the *cut-off* test is more complicated than without a heap. Performing the *cut-off* with a heap would require that one goes through the whole tree and this can not be done in less than $\log(|\mathcal{L}|)$ asymptotically. We therefore do not perform this test, we rather leave all elements in the list. The boxes that would have been discarded by the *cut-off* test will never be

considered for further computation. They will remain in the list until their turn to be processed reaches. Since, we only handle boxes $[x]$ for which $\underline{F}([x]) \leq \tilde{f}$, these boxes will be discarded.

2.4.5 Experimental results

We now give some experimental results obtained when applying these strategies on some problems found in the literature. We distinguish two cases. For the first case we use a simple linked list and for the second case we use a heap. In the first column of Table 2.1 and Table 2.2 we have the description of the problem as found in the literature. For convenience, these problems are listed in the appendix of this thesis. In the second column we have the different strategies; **Bf** for *best-first*, **Depth** for *depth-first*, **Reject1** for the *reject-index* **Reject2** for the *relative reject-index* and **Rejecth** for the hybrid strategy, i.e. the *best-first* combined with the *depth-first*, where we switch to depth-first as soon as the list $\mathcal{L}$ starts to decrease.
In the third column we have the number of function evaluations (Feval), in the fourth, the number of gradient evaluations (Geval), in the fifth the number of Hessian evaluations (Heval). We have in the sixth column the maximal length reached the by list. The last two columns give the number of iterations and the CPU time required by the algorithm. For each problem, the best value according to the metrics (Feval, Geval ...) listed above is written in bold. These experimental results were carried out on a Pentium-IV machine (2.8 Ghz and 1 Gbyte) under the Linux operating system. The time unit is the second.

Tab. 2.1: Performance of the algorithm using a simple linked list

Problem	Strategy	Feval	Geval	Heval	Length	Iteration	Time
	Bf	**330**	286	**108**	**5**	44	**0.01**
	Depth	1552	1380	519	8	223	0.05
RO	Reject1	329	**285**	**108**	6	**44**	**0.01**
	Reject2	329	**285**	**108**	6	**44**	**0.01**
	Rejecth	329	**285**	**108**	6	**44**	**0.01**
	Bf	**2799**	**2069**	**630**	123	**510**	0.21
	Depth	3068	2268	690	**10**	558	**0.18**
SHCB	Reject1	2799	**2069**	**630**	105	**510**	**0.18**
	Reject2	2799	**2069**	**630**	28	**510**	**0.18**
	Rejecth	2822	2092	639	42	513	0.19
	Bf	10756	**8986**	**3278**	290	1567	1.00
	Depth	**10738**	**8986**	**3278**	**20**	**1566**	0.95
GP	Reject1	10756	**8986**	**3278**	375	1567	0.97
	Reject2	10756	**8986**	**3278**	68	1567	0.94
	Rejecth	10756	**8986**	**3278**	131	1567	**0.9**
	Bf	**1504**	**1186**	**397**	58	**249**	0.186
	Depth	1508	1190	**397**	60	**249**	0.186
R4	Reject1	**1504**	**1186**	**397**	60	**249**	0.186
	Reject2	**1504**	**1186**	**397**	**16**	**249**	**0.16**
	Rejecth	**1504**	**1186**	**397**	57	**249**	**0.16**
	Bf	**510**	**350**	**92**	30	**83**	0.69
	Depth	148381	99257	28202	**22**	28192	205
L12	Reject1	519	357	94	31	85	0.7
	Reject2	516	354	93	31	84	0.7
	Rejecth	**510**	**350**	**92**	52	**83**	**0.68**
	Bf	**511**	**347**	**91**	**9**	**90**	**0.47**
	Depth	512	348	92	**9**	**90**	0.49
R8	Reject1	535	371	99	**9**	98	0.51
	Reject2	535	371	99	**9**	98	0.51
	Rejecth	535	371	99	**9**	98	0.5
	Bf	605	409	125	**1**	106	0.22
	Depth	**568**	**372**	**107**	**1**	**100**	**0.21**
G7	Reject1	605	409	125	**1**	106	0.22
	Reject2	605	409	125	**1**	106	0.23
	Rejecth	605	409	125	**1**	106	0.23
	Bf	**1276**	**1206**	**472**	37	**182**	0.23
	Depth	1277	1207	473	**11**	**182**	0.23
JS	Reject1	1443	1349	518	22	216	0.25
	Reject2	1443	1349	518	13	216	0.25
	Rejecth	**1276**	**1206**	**472**	29	**182**	**0.22**
	Bf	**2283**	**1651**	**488**	72	**422**	**1.43**
	Depth	4795	3399	972	**38**	923	2.82
H6	Reject1	2385	1721	506	66	440	1.46
	Reject2	2831	2045	595	57	530	1.72
	Rejecth	2828	2042	595	57	529	1.75
	Bf	**1143**	**813**	**259**	9	**194**	0.28
	Depth	1144	814	**259**	11	**194**	**0.27**
Sch27	Reject1	**1143**	**813**	**259**	9	**194**	0.28

continue on the next page

continued from previous page

Problem	Strategy	Feval	Geval	Heval	Length	Iteration	Time
	Reject2	**1143**	**813**	**259**	9	**194**	0.28
	Rejecth	**1143**	**813**	**259**	**8**	**194**	0.28
	Bf	8270	7724	3162	18	985	0.88
	Depth	8270	7724	3162	18	985	**0.87**
SW	Reject1	8270	7724	3162	18	985	0.88
	Reject2	8270	7724	3162	18	985	**0.87**
	Rejecth	8270	7724	3162	**17**	985	1.02
	Bf	3484336	3483754	1452327	32	338706	86.04
	Depth	3484337	3483755	1452328	32	338706	88.0
INF1	Reject1	3484336	3483754	1452327	32	338706	84.75
	Reject2	3484336	3483754	1452327	32	338706	85.84
	Rejecth	3484336	3483734	1452327	31	338706	87.58
	Bf	5847	4175	**1235**	555	1408	**0.64**
	Depth	7159	5089	1585	**20**	1258	1.02
L3	Reject1	**5662**	**3984**	1238	201	996	**0.64**
	Reject2	10812	7794	2454	65	1885	1.22
	Rejecth	5722	4050	**1235**	330	**993**	0.67
	Bf	**184052**	**127390**	**39344**	9085	**32628**	31.12
	Depth	195250	136334	42111	**21**	34340	22.25
L3*	Reject1	184313	127575	39382	7931	32666	29.59
	Reject2	184096	127416	39373	362	32645	**21.4**
	Rejecth	**184052**	**127390**	**39344**	2703	**32628**	21.71
	Bf	**9040**	**5908**	**1752**	522	**1647**	1.22
	Depth	9962	6666	1940	**11**	1789	1.2
HM2	Reject1	9148	6006	1780	452	1660	1.12
	Reject2	9801	6511	1946	108	1777	1.1
	Rejecth	**9040**	**5908**	**1752**	504	**1647**	**1.03**
	Bf	**237132**	**158148**	**46254**	12258	**42662**	54.49
	Depth	238499	158483	47041	**19**	43270	27.4
HM2*	Reject1	237500	158476	46355	11156	42715	50.28
	Reject2	239020	159580	46700	470	42991	**27.04**
	Rejecth	**237132**	**158148**	**46254**	10313	**42662**	35.63
	Bf	37430	23686	**6922**	2218	**6897**	8.54
	Depth	67784	43620	12385	**17**	12289	14
HM3	Reject1	37398	**23652**	6923	2006	6898	8.6
	Reject2	711434	48004	13837	783	12966	16.02
	Rejecth	**37397**	23653	**6922**	1851	**6897**	**8.34**
	Bf	268051	**170663**	**49641**	15525	**49225**	101.32
	Depth	310180	200338	58017	**22**	57136	65
HM3*	Reject1	268210	170822	49666	14336	49234	97.933
	Reject2	271220	172956	50275	3169	49785	**60.38**
	Rejecth	**268045**	170657	**49641**	12036	**49225**	74.23
	Bf	**917956**	**915394**	**384049**	28958	**108166**	943.24
	Depth	1155549	1152641	479359	**25**	141326	626.89
KOW	Reject1	976538	973962	407867	6643	115864	567.35
	Reject2	983984	981406	410773	999	117074	556.87
	Rejecth	986014	983362	411556	999	117373	**542.19**
	Bf	16686	15708	6228	541	**2038**	2.07
	Depth	16972	15994	6336	**12**	2075	1.96
WK	Reject1	**16680**	**15702**	**6225**	657	**2038**	1.99

continue on the next page

continued from previous page							
Problem	Strategy	Feval	Geval	Heval	Length	Iteration	Time
	Reject2	16935	15957	6321	42	2071	**1.93**
	Rejecth	16935	15957	6321	548	2071	1.97

Tab. 2.2: Performance of the algorithm using the heap

Problem	Strategy	Feval	Geval	Heval	Length	Iteration	Time
	Bf	336	292	108	24	64	**0.01**
	Depth	1979	1849	714	**12**	279	0.09
RO	Reject1	740	718	288	22	105	0.02
	Reject2	1223	1159	457	19	171	0.05
	Rejecth	**311**	**289**	**114**	29	**34**	**0.01**
	Bf	2802	2072	630	125	513	0.17
	Depth	3008	2224	680	13	548	0.19
SHCB	Reject1	**613**	**565**	**209**	24	**90**	**0.05**
	Reject2	2799	2069	630	28	510	0.18
	Rejecth	671	605	226	**10**	95	**0.05**
	Bf	10760	8989	3278	290	1567	**0.96**
	Depth	**10756**	8986	3278	332	1567	1.06
GP	Reject1	**10756**	8986	3278	375	1567	1.00
	Reject2	**10756**	8986	3278	190	1567	0.97
	Rejecth	**10756**	8986	3278	**93**	1567	**0.96**
	Bf	1512	1200	**398**	74	268	0.07
	Depth	**1504**	**1186**	**397**	**11**	**249**	**0.06**
R4	Reject1	**1504**	**1186**	**397**	60	**249**	**0.06**
	Reject2	**1504**	**1186**	**397**	53	**249**	**0.06**
	Rejecth	**1504**	**1186**	**397**	53	**249**	**0.06**
	Bf	514	353	93	53	**135**	**0.69**
	Depth	148380	99256	28201	**22**	28192	206.7
L12	Reject1	519	357	94	33	138	0.7
	Reject2	516	354	93	39	138	0.7
	Rejecth	**510**	**350**	**92**	52	136	**0.69**
	Bf	515	350	92	41	122	**0.55**
	Depth	601	437	121	**2**	**120**	0.69
R8	Reject1	535	371	99	18	122	0.59
	Reject2	589	425	117	5	122	0.66
	Rejecth	**511**	**347**	**91**	40	122	**0.55**
	Bf	569	372	107	1	178	0.21
	Depth	568	372	107	1	100	0.21
G7	Reject1	568	372	107	1	100	0.21
	Reject2	568	372	107	1	100	0.21
	Rejecth	568	372	107	1	100	0.21
	Bf	**1280**	**1209**	**473**	45	**190**	**0.22**
	Depth	1373	1289	499	**11**	201	0.24
JS	Reject1	1443	1349	518	23	219	0.25
	Reject2	1335	1251	486	19	194	0.23
	Rejecth	1301	1231	482	26	188	.23
	Bf	2297	1664	489	115	468	**1.38**
	Depth	4478	3386	969	37	919	2.86
H6	Reject1	2385	1721	506	78	476	1.46
	Reject2	3459	2445	712	**30**	668	2.08
	Rejecth	**2283**	**1621**	**488**	93	**444**	1.4
	Bf	1143	813	259	11	194	0.29
	Depth	1143	813	259	**10**	194	0.33
SCH27	Reject1	1143	813	259	15	194	**0.28**

continue on the next page

continued from previous page

Problem	Strategy	Feval	Geval	Heval	Length	Iteration	Time
	Reject2	1143	813	259	**10**	194	**0.28**
	Rejecth	1143	813	259	**10**	194	**0.28**
	Bf	8270	7724	3162	27	985	0.89
	Depth	8270	7724	3162	**18**	985	0.87
SW	Reject1	8270	7724	3162	40	985	**0.85**
	Reject2	8270	7724	3162	19	985	0.94
	Rejecth	8270	7724	3162	31	985	0.88
	Bf	3484336	3843754	1452327	410	338706	105.05
	Depth	3484336	3483754	1452327	76	338706	92.03
INF1	Reject1	3484336	3483754	1452327	76	338706	92.03
	Reject2	2484336	3483754	1452327	76	338706	**90.46**
	Rejecth	3484336	3483734	1452377	**31**	338706	91.86
	Bf	9082	5949	1753	589	1724	**0.99**
	Depth	9762	6402	1899	**16**	1776	1.1
HM2	Reject1	**9036**	**5904**	**1752**	475	**1649**	1.01
	Reject2	9424	6150	1824	88	1719	1.27
	Rejecth	9040	5908	**1752**	504	1654	1.01
	Bf	237137	158152	46255	12376	42779	**26.16**
	Depth	239987	160269	46430	11126	42791	27.12
HM2*	Reject1	237655	158657	46430	11126	42791	27.12
	Reject2	237246	158218	46276	**4596**	42684	32.57
	Rejecth	**237132**	**158148**	**46254**	10315	**42662**	27.12
	Bf	5850	4177	1236	555	1408	**0.63**
	Depth	7014	4988	1552	**20**	1235	0.79
L3	Reject1	**5657**	**3981**	1237	204	**1006**	0.64
	Reject2	6280	4470	1393	66	1114	0.72
	Rejecth	5722	4050	**1235**	331	1117	0.66
	Bf	184061	127398	39345	9218	32763	**20.75**
	Depth	184563	127693	**39344**	8328	**32628**	21.84
L3*	Reject1	**183992**	**127330**	**39344**	8328	**32628**	21.84
	Reject2	**183992**	**127330**	**39344**	3397	**32628**	21.44
	Rejecth	184000	127338	**39344**	**1145**	32664	20.89
	Bf	37479	23734	6923	2311	7090	**7.58**
	Depth	48134	31324	9095	**30**	8803	10.13
HM3	Reject1	37416	23668	6924	1980	6948	7.88
	Reject2	38755	24517	7172	96	7147	8.38
	Rejecth	**37397**	**23653**	**6922**	1849	**6909**	7.91
	Bf	268170	170781	49642	15720	49444	54.83
	Depth	269164	171416	49852	**31**	49424	56.71
HM3*	Reject1	268050	170662	**49641**	14531	49235	**56.29**
	Reject2	271094	172598	50195	745	49779	58.3
	Rejecth	**268045**	**170657**	**49641**	11741	**49225**	56.82
	Bf	**919247**	**916685**	**384563**	29693	**109620**	**521.6**
	Depth	1064614	1061916	443428	**60**	128297	577.21
KOW	Reject1	991981	989405	4213704	7090	120122	549.05
	Reject2	1008433	1005877	420077	1043	121669	547.48
	Rejecth	953660	951096	398016	420	113469	583.07
	Bf	17642	16664	6590	544	2158	2.06
	Depth	16972	15994	6336	**15**	2075	1.98
WK	Reject1	**16680**	**15702**	**6225**	657	**2039**	**1.96**

continue on the next page

continued from previous page							
Problem	Strategy	Feval	Geval	Heval	Length	Iteration	Time
	Reject2	16935	15957	6321	42	2071	**1.96**
	Rejecth	16972	15994	6336	539	2075	1.99

The Problems listed above can be grouped in two categories, easy and hard problems. Easy problems are at beginning of the tables and hard problems at the end. Easy problems typically require less that 500 iterations.
We see that for easy problems there is not a great difference between various strategies except for L12, where the *depth-first* is much slower than the other strategies. For these problems the *best-first* and the *reject1* are a bit better than other strategies. For hard problems we see that *reject2*, *depth-first* and the hybrid strategy are better than *best-first* and *reject-index1*. This is because the list became large and the algorithm spends a great part of time handling the list. We see that, in general, *rejecth* realizes a better compromise.
In Table 2.2 the influence of the length of the list is reduced by the use of the heap. We see that in general *depth-first* and *reject2* are worse than other strategies. The *best-first* and *rejecth* show in general the best performances.

To conclude this part, we may say that the hybrid strategy seems to compete with the *best-first* and the *reject-index*, but it is too much problem-dependent. This hybrid strategy could be improved if one could find a better switching metric. Using a heap, the time spent to handle the list is made smaller, and then the *best-first* is to prefer to the other strategies.

Another important point of Algorithm 1 is the way one subdivides the box and the number of parts the box is subdivided into.

2.4.6 Subdivision strategies

In each iteration step, Algorithm 1 computes enclosures for the objective function and enclosures for its gradient and Hessian over the current box, provided f is sufficiently smooth. The smaller a box, the smaller the overestimation of the range of these quantities. It follows that the algorithm is efficient when the box is small. There are two aspects one muss take into consideration when splitting (subdividing) the box, namely, the *direction* or the *coordinate* and the number of subboxes to produce. We first discuss the subdivision direction and then the number of subdivision parts.

Choice of the subdivision direction

In the literature one distinguishes two strategies to find the direction where to split the box.

Strategy A: The largest edge. This is the widely and traditionally used subdivision strategy: It is also the one we assumed when deriving the convergence results in Proposition 2 and 3. The box is subdivided along the coordinate with maximal interval width. It means that the algorithm computes the width

$$w([x]) = (w([x]_1, \ldots, [x]_n))$$

of the box $[x]$ and chooses the coordinate d for which

$$w([x]_d) = \max_{i=1}^{n} w([x]_i).$$

The box is split orthogonally to this direction. The idea with this strategy is that splitting this way leads to boxes with shapes close to a cube, the width of which possibly tends to zero fast. In this way one gets good enclosures of the global minimizers and, hopefully, a good enclosure for the range of the function over the boxes as well. For more details about this strategy see [46].
Now suppose we have the following objective function

$$f(x_1, x_2) = x_1^2 + 1. \tag{2.7}$$

Suppose we want to find its global minimum in the box $[x] = [-2, 3] \times [-2, 3]$. If strategy A is applied, then the box would be split in the second direction x_2 as often as in the first direction . Since this function does not vary in the second direction, splitting in this direction would not help in the search for the minimum. This remark leads to a refined subdivision strategy developed by Hansen [16].

Strategy B: Where the function varies the most. This strategy is described in details in Hansen's book [16], paragraph 9.13. Here we give an overview.

Let

$$f_i(t) = f(x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots, x_n),\ i = 1, \ldots, n,$$

where $x_j = m([x_j])$. As a measure of how much f varies as x_i varies over $[x]_i$, we could use

$$W_i = \max_{t \in [x]_i} f_i(t) - \min_{t \in [x]_i} f_i(t).$$

It would be reasonable to split along the jth component of $[x]$ where j is chosen such that $W_j = \max\{W_i,\ i = 1, \ldots, n\}$. Since determining W_i is another optimization problem, it would be expensive to try to find this quantity exactly. But since (W_i) can be estimated by

$$W_i \leq w(F_i'([x])).w([x]_i),$$

which immediately follows from the mean value form, one can use the quantity in the right side of the inequality to determine the coordinate to split.
Strategy B thus defines the metric function

$$d_i = w(F_i'([x])).w([x]_i)$$

and it chooses the direction d where this quantity is maximal, i.e.

$$w(F_d'([x])).w([x]_d) = \max_{i=1}^{n} w(F_i'([x])).w([x]_i).$$

The box is split in the direction d. Strategy B requires the computation of the gradient, but this value is already available if one uses accelerating devices we will present later. In the definition of d_i the width could be replaced by the absolute value, in this case one has

$$d_i = |F_i'([x])|.w([x]_i).$$

Now the direction where the function is steepest is chosen.
Using Strategy B on the example above one sees that, since $F_2'([x]_1, [x]_2) = 0$, the first direction will almost always be favored for splitting subboxes.
Next, we present strategies for the number of subdivision parts.

Number of subdivision parts

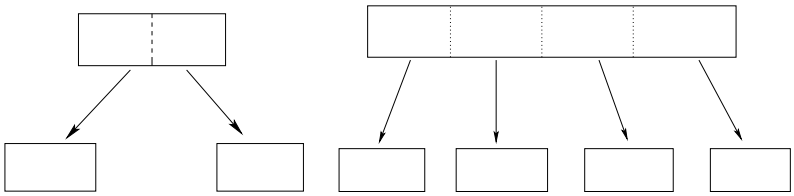

Fig. 2.2: bisection (left) and multiselection (right)

The problem addressed here is to determine the number of times a box should be subdivided at each iteration. If a box is split once then one obtains a *bisection*. If the box is split more than once during an iteration then one is doing a *multisection*. Figure 2.2 shows an example of bisection and of multisection.
The idea of multisection originates from the parallelization of methods for global optimization. Since multisection generates many boxes, this is advantageous to avoid idle processors, i.e. processors having no boxes to work on. This strategy is now used in the serial case too. Suppose that during an iteration the algorithm splits the current box $[x]$ into $[x]_1, \ldots, [x]_n$. Then progress is made if

some of the new boxes $[x]_i$ can be discarded due to the cut-off test or if $[x]_1$ is discarded or contracted due to an accelerating device. Just as the cut-off test the accelerating devices will be more likely to work if the interval extension of the functions used lead only to an only small overestimation of the respective ranges. But overestimation gets smaller as boxes become smaller, this is an advantage of making boxes small rapidly which is achieved by multiselection. The following algorithms (Algorithm 3 and 2 describe our subdivision strategy in detail. Algorithm 2 determines l, not necessarily different, directions along which the boxes will be subdivided in Algorithm 3. If $l = 1$ it is a bisection, if $l > 1$ it is a multisection. If we choose l too large, we generate too many new boxes. Many authors choose $l = 3$. We choose in this work $l = 2$ because we allow some accelerating devices to split boxes, see Section 2.6 for exmaple.

algorithm 2 Determine directions for subdivision

Input: [**x**] the box to subdivide, l the number of components
Output: **List** List of subboxes

compute (d_i), $i = 1, \ldots, n$, according to the Strategy A or Strategy B
for i = 1 to l **do** {determine the directions $r_1, \ldots, r_l$}
  $r_i = \min\{j \in \{1, \ldots, n\} : d_j = \max_{t=1}^{n} d_t\}$;
  $d_{r_i} = d_{r_i}/2$;
end for
List = Split($[x], l, r_1, \ldots, r_l$)

algorithm 3 Split

Input: the box [**y**], number of directions l and directions $r_1, \ldots, r_l$
Output: **List** List of subboxes

$[u] = [y]$,
$[v] = [y]$,
$[u]_{r_l} = [\underline{[y]}_{r_l}\ m([y]_{r_l})]$, {bisection along direction r_l}
$[v]_{r_l} = [m([y]_{r_l}),\ \overline{[y]}_{r_l}]$, {bisection along direction r_l}
if $l = 1$ **then**
  List = $[u] \cup [v]$;
else
  List = Split($[u]$, $l - 1$, $r_1, \ldots, r_{l-1}$) $\cup$ Split($[v]$, $l - 1$, $r_1, \ldots, r_{l-1}$)
end if

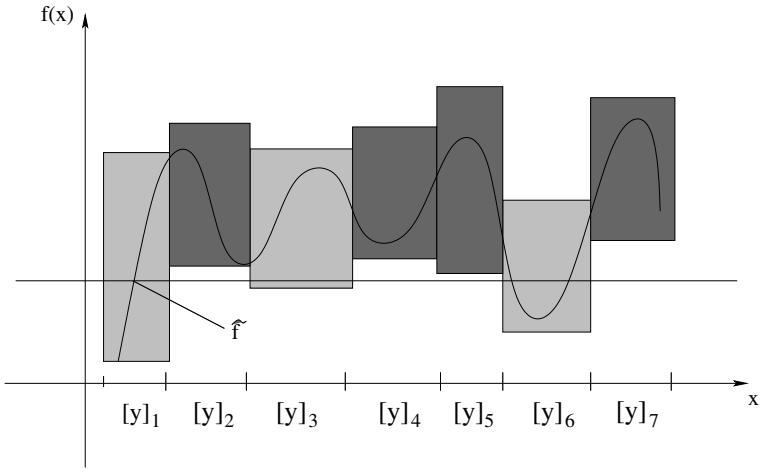

Fig. 2.3: Illustration of the midpoint test

2.5 Accelerating devices

We understand by accelerating devices all techniques that can speed up the convergence of Algorithm 1. In the literature there exist accelerating devices based particularly on the smoothness of the objective function. In the sequel we give an overview of the accelerating devices according to the smoothness of the objective function, see [46, 16, 26] for more details.

2.5.1 The cut_off test

This test is applied to determine boxes that cannot contain the global minimum. In fact, this test finds boxes $[x]$ in the working list for which $\underline{F}([x]) > \tilde{f}$; where $\tilde{f}$ is the upper bound of the minimum known so far. Since for these boxes we have

$$\underline{F}([x]) > \tilde{f} \geq f^* = \min_{x \in [x]} f(x),$$

they cannot contain a global minimizer and need not be considered further. Figure 2.3 illustrates this test. In this case the intervals $[y]_2, [y]_4, [y]_5, [y]_7$ should be discarded. This is one of the cheapest accelerating devices since it requires no extra computation.

2.5.2 Finding a lower function value

The aim of doing this search is to improve the value of $\tilde{f}$ so that the tests like the cut-off test based on the knowledge of this value should be more efficient. The minimum requirement here is that f, the objective function is continuous.

- If f is not differentiable: One can do a *grid or line search.*

- If f is a C^1 function: There exists many methods in this case. One can apply the *steepest descent method* or the *conjugate gradient method*, for example to obtain a lower function value.

- If f is a C^2 function: In this case Newton-like methods to approximate a zero of f' are favored.

we apply the Newton-like method to do the local search since functions we deal with are twice differentiable.

2.5.3 The monotonicity test

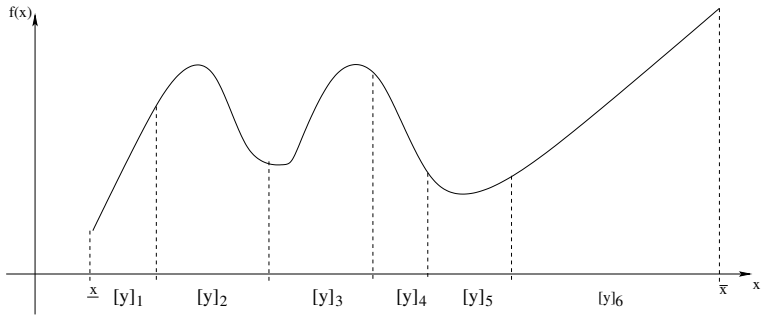

Fig. 2.4: Illustration of the monotonicity test

The monotonicity test determines whether the function f is *strictly monotone* on the current box $[y] \subset [x]^0$. If f is strictly monotone in $[y]$, then $[y]$ cannot contain a global minimum in its interior. Therefore, the algorithm deletes all boxes which satisfies

$$0 \notin F'([y]), \text{ i.e. } 0 \notin F_i' \text{ for an } i \in \{1, \ldots, n\}$$

with the exception of the boundary points of $[x]$ if those are also boundary points of the starting box. Figure 2.4 illustrates this test. Here the function is strictly monotone on the intervals $[y]_1$, $[y]_4$ and $[y]_6$ but only $[y]_4$ should be discarded since $\underline{y}_1 = \underline{x}$ and $\overline{y}_6 = \overline{x}$.

2.5.4 The convexity test

For this test it is assumed that the function is in C^2. This test examines whether the function is convex. If the function is not convex in a subbox $[y] \subset [x]$, then $[y]$ cannot contain a global minimizer in its interior. A necessary and sufficient condition for a function f to be convex is that its hessian matrix should be

positive semidefinite, and this requires the elements on the diagonal to be all nonnegative. Therefore, this test checks for a box $[y]$ whether

$$\overline{F''}([y])_{ii} < 0$$

for some $i \in \{1, \ldots, n\}$. If this is the case, $[y]$ is discarded, with exception of its boundary points if these are also boundary points of the starting box. Figure 2.5 demonstrates the convexity test. Here the function is not convex in $[y]_1$, $[y]_3$ and $[y]_4$, but since $\overline{y}_4 = \overline{x}$, $[y]_4$ can be reduced to $\overline{x}$.

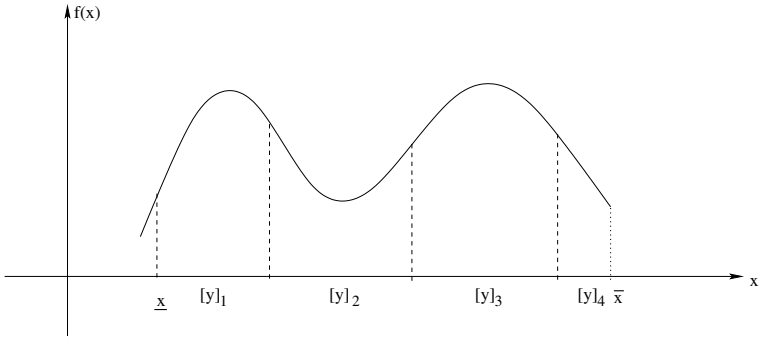

Fig. 2.5: Illustration of the convexity test

2.5.5 The Interval Newton Step

One way to solve the global optimization problem could be to first determine all stationary points, i.e to solve the system of equations

$$f'(x) = 0, \quad x \in [x^0]. \tag{2.8}$$

One can then evaluate f over all stationary points of the function and choose the smallest. The interval Newton step is based on this observation in the sense that it determines a part of the current box $[x]$ which is guaranteed to contain all stationary points of f in $[x]$.

The interval Newton step is one of the most important accelerating devices. The difference with other discarding tests is that this is not an *all or nothing* test. Applying this test on a box, the box can be deleted, the box can be contracted or the box can be split. This is in contrast to the other tests, where the box is either deleted or it is conserved.

The interval Newton test is also the computationally most expensive test since it requires an interval evaluation of the Hessian matrix and the solution of an interval linear system, etc. For this reason we apply only one iteration of an

interval Newton method to solve the system of equations (2.8). Another reason why we do not iterate until convergence is that the boxes produced at a certain stage of the iteration would be discarded by other less expensive tests. We now give a brief description of that variant of the interval Newton method we use here; for more details one can refer to [16, 11, 26, 34].

Nonlinear systems of equations

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a continuously differentiable function. The problem addressed here is to solve the system of equations $f(x) = 0$ in a box $[x]$. For non interval methods it can sometimes be difficult to find one solution, quite difficult to find all solutions, and often impossible to know whether all solutions have been found. Using interval methods, it is on the other hand easy to compute a collection of subboxes of $[x]$ known to contain all zeros of f. Here we present briefly how this is done.
Let J_f denote the Jacobian matrix of f. From the mean value theorem we have

$$f(c) - f(x^*) = J_f(\xi) \cdot (c - x^*),$$

where

$$J_f(\xi) = (f_1'(\xi_1), \ldots, f_n'(\xi_n))^T,$$
$$x^*,\ c \in [x],\ \xi = (\xi_1, \ldots, \xi_n)$$

and

$$\xi_i \in [x] \text{ for } i = 1, \ldots, n.$$

Usually one takes $c = m([x])$. If we assume x^* to be the a zero of f, we get

$$f(c) = J_f(\xi) \cdot (c - x^*). \tag{2.9}$$

If we assume $J_f(\xi) \in \mathbb{R}^{n\times n}$ and all real matrices in an inclusion function $J_f([x]) \in \mathbb{IR}^{n\times n}$ for J_f to be non singular, we have, e.g.

$$\begin{aligned} x^* &= m([x]) - (J_f(\xi))^{-1} \cdot f(m([x])) \\ &\in \underbrace{c - [B] \cdot f(m([x])),}_{N([x])} \end{aligned}$$

where

$$[B] \supseteq \{A^{-1},\ A \in J_f([x])\}.$$

Relation 2.9 is at the basis of many Newton-like interval methods. The vectors $\xi_1, \ldots, \xi_n$ are unknown, but they are contained in the interval vector $[x]$. Therefore, knowing an interval enclosure for $J_f(x)$, $x \in [x]$, we can exploit the

above relation to compute an interval vector containing x^*. One computes an enclosure of the set of solutions x^* of the linear systems

$$f(m([x])) = A \cdot (m([x]) - x^*), \tag{2.10}$$

where

$$A \in \{J_f(\xi),\ \ \xi_i \in [x],\ \ \text{for } i = 1, \ldots, n\}.$$

Usually, an inclusion function for $J_f(x)$ being available, one takes

$$A \in J_f([x]) \in \mathbb{IR}^n$$

since

$$\{J_f(\xi),\ \xi_i \in [x],\ \ \text{for } i = 1, \ldots, n\} \subset J_f([x]).$$

We are therefore interested in computing an interval enclosure for the solution set

$$S = \{x \in [x] :\ \exists A \in J_f([x]).\ \text{s.t. } f(m([x])) = A \cdot (m([x]) - x^*)\}\,,$$

and we know that $x^* \in S$. Abusing notation and terminology we say that an interval vector $[y]$ solves the linear interval system

$$f(m([x])) = J_f([x]) \cdot (m([x]) - x^*)$$

if $[y] \supseteq S$.
As a preconditioner for the systems (2.10), one usually takes the numerically computed inverse $R \in \mathbb{R}^{n \times n}$ of $m(J_f([x]))$. This will also be the choice in this work.
Doing so (2.10) is restated as

$$R \cdot f(m([x])) = R \cdot A \cdot (m([x]) - x^*). \tag{2.11}$$

The idea is that one has

$$\begin{aligned} R \cdot (J_f([x])) &= R \cdot (J_f([x]) - m(J_f([x])) + m(J_f([x]))) \\ &= R \cdot m(J_f([x])) + R \cdot (J_f([x]) - m(J_f([x]))) \\ &\simeq I + R \cdot (J_f([x]) - m(J_f([x]))). \end{aligned}$$

If one sets $b = R \cdot f(m([x]))$, $[A] = R \cdot J_f([x])$ and $c = m([x])$, Equation (2.11) becomes the linear interval systems

$$b = [A] \cdot (c - x^*). \tag{2.12}$$

We apply the *Gauss-Seidel* iteration to solve the system of equations (2.12)

Gauss-Seidel Iteration

We give a brief description taken from book the [11]. For more details about this topic we refer to [16] or [46]. We are interested in the solution set

$$S := \{x \in [x] : \quad A \cdot (c - x) = b, \quad \text{for } A \in [A]\}$$

of the interval linear equation

$$[A] \cdot (c - x) = b.$$

The Gauss-Seidel iteration is obtained by writing the linear system $A \cdot (c-x) = b$ componentwise as

$$\sum_{j=1}^{n} A_{ij} \cdot (c_j - x_j) = b_i, \quad i = 1, \ldots, n, \tag{2.13}$$

and solving the ith equation for the ith variable, assuming that $A_{ii} \neq 0$. Then we have

$$x_i = c_i - (b_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij} \cdot (x_j - c_j))/A_{ii}.$$

Since we are interested in all possible solutions of (2.13) for all $A \in [A]$, $i = 1, \ldots, n$, the inclusion property of interval arithmetic gives

$$x_i \in \underbrace{c_i - (b_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} [A_{ij}] \cdot ([x_j] - c_j))/[A_{ii}]}_{=:[z]_i}, \quad i = 1, \ldots, n, \tag{2.14}$$

provided $0 \notin [A]_{ii}$. The new enclosure $[z]$ for the solution set S can therefore be obtained from $[x]$ by computing the interval vector components $[z]_i$ according to (2.14) yielding

$$S \subseteq [z] \cap [x].$$

The Gauss-Seidel approach uses the fact that it is possible to improve the enclosures $[z]$ since at the ith step improved enclosures $[z_1], \ldots, [z_n]$ are already available. Thus, we compute

$$[y]_i := \left(c_i - \left(b_i + \frac{\sum_{j=1}^{i-1} [A]_{ij} \cdot ([x]_j - c_j) + \sum_{j=i+1}^{n} [A]_{ij} \cdot ([x]_j - c_j)}{[A]_{ii}} \right) \right) \cap [x]_i \tag{2.15}$$

and we get

$$N_{GS}([x]) := [y].$$

Accordingly, we have

$$S \subseteq N_{GS}([x]) \subseteq [z] \cap [x],$$

and turning back to our initial definition of $[A]$, b and c, we know that every zero of f lying in $[x]$ also lies in $N_{GS}([x])$. The interval Newton Gauss-Seidel iteration starts with an interval vector $[x]^{(0)}$ and iterates according to

$$[x]^{k+1} := N_{GS}([x]^{(k)}), \quad k = 0, 1, 2, \ldots.$$

The intersection performed in (2.15) prevents the method form diverging. If an empty intersection occurs it means that the function f has no zero in $[x]$.
If $0 \in [A]_{ii}$ for some i, then the extended interval arithmetic presented in Section 1.1 is applied. In this case the division could produce two intervals and the iteration should continue with these two intervals, details will be given in Algorithm 4.
The following theorem summarize the properties of the iteration (2.15).

Theorem 9 *Let $f : D \subseteq \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function and let $[x] \in \mathbb{IR}^n$ be an interval vector with $[x] \subseteq D$. Then $N_{GS}([x])$ defined by (2.15) has the following properties:*

1. *Every zero $x^* \in [x]$ of f satisfies $x^* \in N_{GS}([x])$.*
2. *If $N_{GS}([x]) = \emptyset$, then there exists no zero of f in $[x]$.*
3. *If $N_{GS}([x]) \subset \overset{\circ}{[x]}$, then there exists a unique zero of f in $[x]$ and hence in $N_{GS}([x])$.*

For a proof see [16].
The way in which $[y]_i$ is computed in (2.15) distinguishes the various interval Newton methods. So also does the way in which $[A]$ is defined. There exist many variations of interval Newton methods among which the Krawczyk's method, and the Hansen-Greenberg's method, see [46, 16]. We now formulate an algorithm implementing one step of the Gauss-Seidel iteration.
Algorithm 4 first performs the single component steps of the Gauss-Seidel iterations for all i with $0 \notin [A]_{ii}$ and then for the remaining indices with $0 \in [A]_{ii}$. Using this strategy, it is possible that the interval $[y]_i$ become smaller by intersections with the old values $[y]_i$ before the first splitting, due to $0 \in [A_{ii}]$, is produced.

algorithm 4 NewtonStep

Input: f**:** the function, $[y]$ **:** the box, J_f: inclusion function for the jacobian.
Output: p the number of resulting subboxes, V the set of subboxes.

$c = m([y])$; {compute the midpoint of $[y]$}
$R = (m(J_f([y])))^{-1}$ {inverse of the midpoint matrix as preconditioner }
if $m(J_f([x]))$ singular **then**
  $R = I$ {R equals the identity}
end if
$[A] = R \cdot J_f([y])$; $[b] = R \cdot f(c)$; $[y_c] = [y] - c$; $p = 0$;
for $i = 1$ to n **do**
  if $0 \notin [A_{ii}]$ **then**

$$[y]_i := (c_i - (b_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} [A]_{ij} \cdot ([y]_j - c_j))/\,[A]_{ii}) \cap [y]_i$$

    if $[y]_i = \emptyset$ **then**
      $p = 0$; **return** ; {no solution in $[y]$}
    end if
    $[y_c]_i = [y]_i - c_i$;
  end if
end for
for $i = 1$ to n **do**
  if $0 \in [A_{ii}]$ **then**

$$[z] := (c_i - (b_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} [A]_{ij} \cdot ([y]_j - c_j))/[A]_{ii}) \cap [y]_i; \{[z] = [z]_1 \cup [z]_2\}$$

    if $[z] = \emptyset$ **then**
      $p = 0$; **return** {no solution in $[y]$}
    end if
    $[y]_i = [z]_1$; $[y_c]_i = [y]_i - c_i$;
    if $[z]_2 \neq \emptyset$ **then**
      $p = p + 1$; $[V]_p = [y]$; $[V]_{pi} = [z]_2$; { store part of $[y] \in [V]_p$}
    end if
  end if
end for
$p = p + 1$; $[V]_p = [y]$;
return $[V]$, p;

2.5.6 Position of accelerating devices

Many of the values computed during one step of Algorithm 1 can be reused during other steps, therefore the position of each step in the algorithm is of particular importance.
If one uses the first order Taylor form as the inclusion function of the objective function, then the enclosure of the gradient can be reused to perform the monotonicity test. If the second order Taylor form is used, then the enclosure of the Hessian matrix can be reused in the concavity test. The interval Newton iteration is placed after the convexity test, so that the Hessian matrix can be reused.
We present in the next section a new accelerating device based on the one-dimensional Newton iteration.

2.6 A new accelerating device

The aim of this accelerating device is to remove parts of boxes which cannot contain global minimizers.
We try to compute an interval vector containing the set

$$S = \{x \in [x] \ : \ f(x) \in [f^*]\}, \tag{2.16}$$

where $[f^*]$ is an interval known to contain the global minimum. For example, we can take

$$[f^*] = (-\infty, \ \tilde{f}]$$

or

$$[f^*] = [\underline{f}, \ \tilde{f}],$$

where $\tilde{f}$ is the so far known best upper bound for the global minimum and $\underline{f}$ is a lower bound. One can take, for example,

$$\underline{f} = \min_{[y] \in \mathcal{L}} \underline{F}([y])$$

or

$$\underline{f} = \underline{F}([x]^0).$$

The idea behind our new accelerating device is rather simple. If S is not empty, then there exists $x \in [x]$ and $\hat{f} \in [f^*]$ such that

$$g(x) := f(x) - \hat{f} = 0.$$

Knowing the enclosure function $G([x]) = F([x]) - [f^*]$ for g, we now apply a Newton-like iteration to find a zero of g.

2.6.1 A method based on a 1-dimensional Newton step

Let $f : B \subseteq \mathbb{R}^n \to \mathbb{IR}$ be differentiable, let $[f^*]$ be an interval. Assume that there is some $\hat{x}$ with

$$f(\hat{x}) = \hat{f} \in [f^*].$$

Let m denote any fixed vector in $[x]$, for example $m = m([x])$. Moreover, let $i \in \{1, \ldots, n\}$ be a fixed direction. Then, by the mean value theorem

$$f(\hat{x}) - f(\hat{x}_1, \ldots, \hat{x}_{i-1}, m_i, \hat{x}_{i+1}, \ldots, \hat{x}_n) = \frac{\partial f(\xi)}{\partial x_i} \cdot (\hat{x}_i - m_i), \tag{2.17}$$

where

$$\xi = (\hat{x}_1, \ldots, \hat{x}_{i-1}, \xi_i, \hat{x}_{i+1}, \ldots, \hat{x}_n), \quad \xi_i \in [x_i].$$

If $[g_i]$ is an interval such that

$$[g_i] \supseteq \left\{ \frac{\partial f(y)}{\partial x_i},\ y \in [x] \right\}$$

we have, solving (2.17) for $\hat{x}_i$

$$\begin{aligned} \hat{x}_i &= m_i - \frac{f(\hat{x}_1, \ldots, \hat{x}_{i-1}, m_i, \hat{x}_{i+1}, \ldots, \hat{x}_n) - f(\hat{x})}{\frac{\partial f(\xi)}{\partial x_i}} \\ &\in m_i - \frac{f(\hat{x}_1, \ldots, \hat{x}_{i-1}, m_i, \hat{x}_{i+1}, \ldots, \hat{x}_n) - f(\hat{x})}{[g_i]}. \end{aligned}$$

If F is an inclusion function for f, then we can use

$$f(\hat{x}_1, \ldots, \hat{x}_{i-1}, m_i, \hat{x}_{i+1}, \ldots, \hat{x}_n) \in F([x_1], \ldots, [x_{i-1}], m_i, [x_{i+1}], \ldots, [x_n])$$

and

$$f(\hat{x}) \in [f^*]$$

to obtain

$$\hat{x}_i \in m_i - \frac{F([x_1], \ldots, [x_{i-1}], m_i, [x_{i+1}], \ldots, [x_n]) - [f^*]}{[g_i]}.$$

We thus have proved the following lemma.

Lemma 2 *Let $f : B \subset \mathbb{R}^n \to \mathbb{R}$ be differentiable, let $[x] \in \mathbb{IR}^n$, $[x] \subseteq B$ and $[f^*] \in \mathbb{IR}$. Consider*

$$S = \{x \in [x] \ :\ f(x) \in [f^*]\}.$$

If i is any direction, $i \in \{1, \ldots, n\}$ then

$$S \subseteq ([x_1], \ldots, [x_{i-1}], [\hat{x}_i], [x_{i+1}], \ldots, [x_n]),$$

with

$$[\hat{x}_i] = \left\{ m_i - \frac{F([x_1], \ldots, [x_{i-1}], m_i, [x_{i+1}], \ldots, [x_n]) - [f^*]}{[g_i]} \right\} \cap [x_i],$$

where

$$F([x_1], \ldots, [x_{i-1}], m_i, [x_{i+1}], \ldots, [x_n])$$

is an inclusion function for

$$f(x_1, \ldots, x_{i-1}, m_i, x_{i+1}, \ldots, x_n)$$

and $[g_i]$ is an interval containing the i-th partial derivatives,

$$[g_i] \supseteq \left\{ \frac{\partial f(y)}{\partial x_i},\ y \in [x] \right\}.$$

The result of this lemma yields the following procedure. Here, we use the notation $G(x)$ to denote the function

$$\begin{array}{l} G:\ [x_i] \longrightarrow \mathbb{IR}, \\ G(t) = F([x_1], \ldots, [x_{i-1}], t, [x_{i+1}], \ldots, [x_n]) - [f^*]. \end{array} \tag{2.18}$$

Procedure

1. choose a direction i, $1 \leq i \leq n$.
2. apply one step of a *formal* interval Newton method to G from (2.18). i.e.

$$[\hat{x}_i] = \left\{ m_i - \frac{G(m_i)}{[g_i]} \right\} \cap [x_i], \tag{2.19}$$

   where $m_i \in [x_i]$ and $[g_i]$ is an interval containing

$$\frac{\partial f(x)}{\partial x_i} \text{ for all } x \in [x].$$

3. replace $[x_i]$ by $[\hat{x}_i]$

Note that we can take $[g_i] = F_i'([x])$, the i-th component of the inclusion function of the gradient of f, a quantity which has usually already been computed in the global optimization algorithm in order to perform the monotonicity test, for example. Thus the Newton step above is indeed cheap computationally.
The following situations can occur doing this iteration:

1. The intersection is empty, meaning that there is no $x \in [x]$ with $f(x) \in [f^*]$. Consequently, the whole box $[x]$ should no longer be considered for the search of minimum points (global minimizers).

2. From an interval $[x_i]$ one obtains an interval $[\hat{x}_i]$ with $[\hat{x}_i] \subset [x]_i$. In this case the search should continue with the smaller box

$$[x]' = ([x_1], \ldots, [x_{i-1}], [\hat{x}_i], [x_{i+1}], \ldots, [x_n]) \subset [x].$$

3. From $[x_i]$, due to extended interval arithmetic, one obtains two intervals $[\hat{x}_{i_1}]$ and $[\hat{x}_{i_2}]$ with $[\hat{x}_{i_1}] \subset [x_i]$ and $[\hat{x}_{i_2}] \subset [x_i]$. In this case the search continues with these two boxes,

$$\begin{aligned} [x]' &= ([x_1], \ldots, [x_{i-1}], [x_{i_1}], [x_{i+1}], \ldots, [x_n]) \subset [x] \\ [x]'' &= ([x_1], \ldots, [x_{i-1}], [x_{i_2}], [x_{i+1}], \ldots, [x_n]) \subset [x]. \end{aligned}$$

4. From the interval $[x_i]$ one obtains $[\hat{x}_i]$ as the interval $[\hat{x}_i] = [x_i]$. This is the unfavorable case since we have no improvement. One should try to avoid this situation for example by choosing an appropriate direction i as we will discuss later.

Before we discuss some properties of (2.19), let's take some examples.

Example 5 $f(x) = x_1^2 - x_1 + x_2^2 - 1,\ [x_1] = [x_2] = [1, 3],\ [f^*] = [-1, 1]$

1 We choose the first direction, i.e. $i = 1$,

2 We obtain the function $G(x_1) = f(x_1, [1, 3]) - [-1, 1] = x_1^2 - x_1 + [-1, 9]$,

3 with $\frac{\partial f(x)}{\partial x_1} = 2x_1 - 1 \in [1, 5] =: [g_1]$ *for* $x_1 \in [x_1]$ *and*

$$m([x_1]) = \frac{1}{2}(\overline{x}_1 + \underline{x}_1) = 2,$$

we obtain

$$G(2) = [1, 11]$$

and

$$[\hat{x}_1] = \left\{2 - \frac{[1, 11]}{[1, 5]}\right\} \cap [1, 3] = [-8, \frac{8}{5}] \cap [1, 3] = [1, \frac{8}{5}].$$

Example 6 $f(x) = x_1^2 - 2x_1 + 2x_2 + 2,\ [x_1] = [x_2] = [0, 2],\ [f^*] = [0, 0]$

1 We choose again $i = 1$,

2 We obtain the function $G(x_1) = f(x_1, [0, 2]) = x_1^2 - 2x_1 + [2, 6]$,

3 with $\frac{\partial f(x)}{\partial x_1} = 2x_1 - 2 \in [-2, 2] =: [g_1]$ *for* $x_1 \in [x_1]$ *and*

$$m([x_1]) = \frac{1}{2}(\overline{x}_1 + \underline{x}_1) = 1,$$

one obtains

$$G(1) = [1, 5]$$

and

$$[\hat{x}_1] = \left\{1 - \frac{G([1, 1])}{[-2, 2]}\right\} \cap [1, 3] = [0, \frac{1}{2}] \cup [\frac{3}{2}, 2].$$

One more iteration with the interval $[0, \frac{1}{2}]$ or $[\frac{3}{2}, 2]$ yields the empty set, proving that there is no point x with $f(x) = 0$ in the box $[0, 2]^2$.

Example 7 $f(x) = e^{x_1} - x_1 \cdot x_2 - 2,\ [x_1] = [x_2] = [-2, 2],\ [f^*] = [-2, 2]$

1 We choose now the second direction, i.e. $i = 2$,

2 We obtain the function
$G(x_2) = f([-2, 2], x_2) = [e^{-2} - 2, e^2 - 2] - x_2 \cdot [-2, 2]$.

3 with $\frac{\partial f(x)}{\partial x_2} = -x_1 \in [-2, 2]$ *for* $x_2 \in [x_2]$ *and*

$$m([x_2]) = \frac{1}{2}(\overline{x}_1 + \underline{x}_1) = 0,$$

one obtains

$$G(0) = [e^{-2} - 2, e^2 - 2]$$

and

$$[\hat{x}_2] = \left\{0 - \frac{[e^{-2} - 2, e^2 - 2]}{[-2, 2]}\right\} \cap [-2, 2] = [-2, 2].$$

Here we obtained no improvement.

We now discuss properties of $[\hat{x}_i]$.

2.6.2 Properties of $[\hat{x}_i]$ in (2.19)

Let's consider equation (2.19)

$$[\hat{x}_i] = \left\{ m_i - \frac{G(m_i)}{[g_i]} \right\} \cap [x_i], \tag{2.20}$$

The aim of this part is to explicitly compute the bounds of $[\hat{x}_i]$. To estimate the improvement obtained, we compare the width of $[\hat{x}_i]$ with the width of $[x_i]$. For the sake of notational simplicity, we put $[g_i] = [g_1, g_2]$ and $m_i = m([x_i]) = m$. We also drop the index i.

case 1: $g_1 > 0$**.** We have to distinguish three cases, since $G(m)$ is an interval and not a point (degenerated interval) as it would be the case in the classic interval Newton iteration.

1. $G(m) > 0$, i.e. $\underline{G}(m) > 0$.

$$\begin{aligned}
[\hat{x}] &= \{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}] \\
&= \left\{ m - \left[\frac{\underline{G}(m)}{g_2}, \frac{\overline{G}(m)}{g_1} \right] \right\} \cap [\underline{x}, \overline{x}] \\
&= \left[m - \frac{\overline{G}(m)}{g_1}, m - \frac{\underline{G}(m)}{g_2} \right] \cap [\underline{x}, \overline{x}] \\
&= \left[\max \left\{ \underline{x}, m - \frac{\overline{G}(m)}{g_1} \right\}, \min \left\{ \overline{x}, m - \frac{\underline{G}(m)}{g_2} \right\} \right] \\
&= \left[\max \left\{ \underline{x}, m - \frac{\overline{G}(m)}{g_1} \right\}, m - \frac{\underline{G}(m)}{g_2} \right].
\end{aligned}$$

This gives

$$w([\hat{x}]) = \begin{cases} \frac{1}{2} w([x]) - \frac{\underline{G}(m)}{g_2}, & \text{if } \underline{x} \geq m - \frac{\overline{G}(m)}{g_1}, \\ \frac{\overline{G}(m)}{g_1} - \frac{\underline{G}(m)}{g_2}, & \text{otherwise.} \end{cases}$$

2. $\overline{G}(m) < 0$, the same development yields

$$[\hat{x}] = \left[m - \frac{\overline{G}(m)}{g_2}, \min \left\{ \overline{x}, m - \frac{\underline{G}(m)}{g_1} \right\} \right]$$

and

$$w([\hat{x}]) = \begin{cases} \frac{1}{2} w([x]) + \frac{\overline{G}(m)}{g_2}, & \text{if } \overline{x} \leq m - \frac{\overline{G}(m)}{g_2}, \\ \frac{\overline{G}(m)}{g_2} - \frac{\underline{G}(m)}{g_1}, & \text{otherwise.} \end{cases}$$

3. $0 \in G(m)$. Then

$$\begin{aligned}[\hat{x}] &= \{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}] \\ &= \{m - [\underline{G}(m), \overline{G}(m)]/[g]\} \cap [\underline{x}, \overline{x}] \\ &= \left[\max\left\{\underline{x}, m - \frac{\overline{G}(m)}{g_1}\right\}, \min\left\{\overline{x}, m - \frac{\underline{G}(m)}{g_1}\right\}\right]\end{aligned}$$

and we get

$$w([\hat{x}]) = \begin{cases} w([x]), & \text{if } \underline{x} \geq m - \frac{\overline{G}(m)}{g_1} \text{ and } \overline{x} \leq m - \frac{\underline{G}(m)}{g_1} \\ \frac{1}{2}(w([x])) + \frac{\overline{G}(m)}{g_1}, & \text{if } \overline{x} \leq m - \frac{\underline{G}(m)}{g_1} \text{ and } \underline{x} \leq m - \frac{\overline{G}(m)}{g_1} \\ \frac{1}{2}(w([x])) - \frac{\underline{G}(m)}{g_1}, & \text{if } \overline{x} \geq m - \frac{\underline{G}(m)}{g_1} \text{ and } \underline{x} \geq m - \frac{\overline{G}(m)}{g_1} \\ \frac{w(G(m))}{g_1}, & \text{if } \overline{x} \leq m - \frac{\underline{G}(m)}{g_1} \text{ and } \underline{x} \geq m - \frac{\overline{G}(m)}{g_1} \end{cases}$$

case 2: $[g] < 0$**, i.e.** $g_2 < 0$. One obtains, proceeding as above, the following results.

1. $\underline{G}(m) > 0$.

$$[\hat{x}] = \left[m - \frac{\underline{G}(m)}{g_1}, \min\left\{\underline{x}, m - \frac{\overline{G}(m)}{g_2}\right\}\right]$$

and

$$w([\hat{x}]) = \begin{cases} \frac{1}{2}w([x]) + \frac{\underline{G}(m)}{g_1}, & \text{if } \underline{x} \leq m - \frac{\overline{G}(m)}{g_2}, \\ \frac{\underline{G}(m)}{g_1} - \frac{\overline{G}(m)}{g_2}, & \text{otherwise.} \end{cases}$$

2. $\overline{G}(m) < 0$.

$$[\hat{x}] = \left[\max\left\{\overline{x}, m - \frac{\underline{G}(m)}{g_2}\right\}, m - \frac{\overline{G}(m)}{g_1}\right].$$

and

$$w([\hat{x}]) = \begin{cases} \frac{1}{2}w([x]) - \frac{\overline{G}(m)}{g_1}, & \text{if } \overline{x} \geq m - \frac{\underline{G}(m)}{g_2}, \\ \frac{\underline{G}(m)}{g_2} - \frac{\overline{G}(m)}{g_1}, & \text{otherwise.} \end{cases}$$

3. $0 \in G(m)$. Then

$$[\hat{x}] = \left[\max\left\{\underline{x}, m - \frac{\overline{G}(m)}{g_2}\right\}, \min\left\{\overline{x}, m - \frac{\underline{G}(m)}{g_2}\right\}\right]$$

and we get

$$w([\hat{x}]) = \begin{cases} w([x]), & \text{if } \underline{x} \geq m - \frac{\overline{G}(m)}{g_2} \text{ and } \overline{x} \leq m - \frac{\underline{G}(m)}{g_2} \\ \frac{1}{2}(w([x])) + \frac{\overline{G}(m)}{g_2}, & \text{if } \overline{x} \leq m - \frac{\underline{G}(m)}{g_2} \text{ and } \underline{x} \leq m - \frac{\overline{G}(m)}{g_2} \\ \frac{1}{2}(w([x])) - \frac{\underline{G}(m)}{g_2}, & \text{if } \overline{x} \geq m - \frac{\underline{G}(m)}{g_2} \text{ and } \underline{x} \geq m - \frac{\overline{G}(m)}{g_2} \\ \frac{w(G(m))}{g_2}, & \text{if } \overline{x} \leq m - \frac{\underline{G}(m)}{g_2} \text{ and } \underline{x} \geq m - \frac{\overline{G}(m)}{g_2} \end{cases}$$

case 3: $0 \in [g]$**, i.e.** $g_1 \leq 0 \leq g_2$**.** This is actually the case we will have in a global optimization algorithm, since otherwise the function is monotone in direction i on the current box and the box should be discarded with exception of its boundary points. Equation (2.19) can be written as

$$[\hat{x}] = N(x,[x]) \cap [x] = \{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}]$$

By the properties of extended interval arithmetic one obtains

- if $0 \in \overset{\circ}{G}(m)$ then $G(m)/[g] = [-\infty, +\infty]$ and $[\hat{x}] = [x]$.
- $\overline{G}(m) \leq 0$ and $g_2 = 0$, then

$$\begin{aligned}
[\hat{x}] &= \{\{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}]\} \\
&= \{m - [\frac{\overline{G}(m)}{g_1}, +\infty]\} \cap [\underline{x}, \overline{x}] \\
&= [\underline{x}, \min\{\overline{x}, m - \frac{\overline{G}(m)}{g_1}\}] \\
&= [\underline{x}, m - \frac{\overline{G}(m)}{g_1}]
\end{aligned}$$

  which gives

$$w([\hat{x}]) = \frac{1}{2}(w([x])) - \frac{\overline{G}(m)}{g_1} \leq \frac{1}{2}(w([x])).$$

- if $\overline{G}(m) \leq 0$ and $g_1 = 0$, then

$$\begin{aligned}
[\hat{x}] &= \{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}] \\
&= \{m - [-\infty, \frac{\overline{G}(m)}{g_2}]\} \cap [\underline{x}, \overline{x}] \\
&= [\max\{\underline{x}, m - \frac{\overline{G}(m)}{g_2}\}, \overline{x}] \\
&= [m - \frac{\overline{G}(m)}{g_2}, \overline{x}],
\end{aligned}$$

  which gives

$$w([\hat{x}]) = \frac{1}{2}(w([x])) + \frac{\overline{G}(m)}{g_2} \leq \frac{1}{2}(w([x])).$$

- if $\underline{G}(m) \geq 0$ and $g_2 = 0$, then

$$\begin{aligned}
[\hat{x}] &= \{\{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}] \\
&= \{m - [-\infty, \frac{\underline{G}(m)}{g_1}]\} \cap [\underline{x}, \overline{x}] \\
&= [\max\{\underline{x}, m - \frac{\underline{G}(m)}{g_1}\}, \overline{x}] \\
&= [m - \frac{\underline{G}(m)}{g_1}, \overline{x}],
\end{aligned}$$

  which gives

$$w([\hat{x}]) = \frac{1}{2}(w([x])) + \frac{\underline{G}(m)}{g_1} \leq \frac{1}{2}(w([x])).$$

- if $\underline{G}(m) \geq 0$ and $g_1 = 0$, then

$$\begin{aligned}
[\hat{x}] &= \{\{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}] \\
&= \{m - [\frac{\underline{G}(m)}{g_2}, +\infty]\} \cap [\underline{x}, \overline{x}] \\
&= [\underline{x}, \min\{\overline{x}, m - \frac{\underline{G}(m)}{g_2}\}] \\
&= [\underline{x}, m - \frac{\underline{G}(m)}{g_2}]
\end{aligned}$$

  which gives

$$w([\hat{x}]) = \frac{1}{2}(w([x])) - \frac{\underline{G}(m)}{g_2} \leq \frac{1}{2}(w([x])).$$

- $\overline{G}(m) \leq 0$ and $g_1 < 0 < g_2$, then

$$\begin{aligned}
[\hat{x}] &= \{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}] \\
&= \{m - ([-\infty, \frac{\overline{G}(m)}{g_2}] \cup [\frac{\overline{G}(m)}{g_1}, \infty])\} \cap [\underline{x}, \overline{x}] \\
&= [\max\{\underline{x}, m - \frac{\overline{G}(m)}{g_2}\}, \overline{x}] \cup [\underline{x}, \min\{\overline{x}, m - \frac{\overline{G}(m)}{g_2}\}] \\
&= [m - \frac{\overline{G}(m)}{g_2}, \overline{x}] \cup [\underline{x}, m - \frac{\overline{G}(m)}{g_1}] = [\hat{x}]' \cup [\hat{x}]''.
\end{aligned}$$

- $\underline{G}(m) \geq 0$, $g_1 < 0 < g_2$

$$\begin{aligned}
[\hat{x}] &= \{\{m - G(m)/[g]\} \cap [\underline{x}, \overline{x}]\} \\
&= \{m - ([-\infty, \frac{\underline{G}(m)}{g_1}] \cup [\frac{\underline{G}(m)}{g_2}, \infty])\} \cap [\underline{x}, \overline{x}]\} \\
&= [\max\{\underline{x}, m - \frac{\underline{G}(m)}{g_1}\}, \overline{x}] \cup [\underline{x}, \min\{\overline{x}, m - \frac{\overline{G}(m)}{g_2}\}] \\
&= [m - \frac{\underline{G}(m)}{g_1}, \overline{x}] \cup [\underline{x}, m - \frac{\underline{G}(m)}{g_2}] = [\hat{x}]' \cup [\hat{x}]''.
\end{aligned}$$

We sum up these computations in the following lemma.

Lemma 3 *Assume that* 0 *is not in the interior of* $G(m)$*, then*

i If 0 *lies on the boundary of* $[g]$*, then* $N(m, [x])$ *is a (possibly empty) interval* $[\hat{x}]$ *with*

$$w([\hat{x}]) \leq \frac{1}{2} w([x]).$$

ii If 0 *lies in the interior of* $[g]$ *and* $0 \notin G(m)$*, then* $N(m, [x])$ *is the union of two (possibly empty) intervals* $[\hat{x}]'$, $[\hat{x}]''$ *and*

$$\begin{aligned}
w\left([\hat{x}]'\right) &< \tfrac{1}{2} w([x]) \\
w([\hat{x}]'') &< \tfrac{1}{2} w([x]).
\end{aligned}$$

In the case that 0 *lies in the interior of* $[g]$ *and* $G(m)$*, then we have*

$$N(m, [x]) = [x].$$

It would be interesting if one could choose a direction i so as to have the quantity $\frac{\underline{G}(m)}{g_1} - \frac{\underline{G}(m)}{g_2}$ or $\frac{\overline{G}(m)}{g_2} - \frac{\overline{G}(m)}{g_1}$ minimized without calculating $G(m(x))$ for each direction. An indicator could estimate this quantity. We don't know so far how to do it.
The aim of this new test is not to iterate the procedure until a an eventual convergence. We simply apply one step of the procedure to discard, contract or split the current box in the global optimization algorithm. In the next section we present the steps of the algorithm with this new test and other discarding tests.

2.6.3 A new algorithm for global optimization problems

The following algorithm enhances the basic Algorithm 1 with the new test *one_newton_test*. We take $[f^*] = [-\underline{F}([x]^0),\ \tilde{f}]$. If the global minimum f^* is known, we can take $[f^*] = [f^*,\ f^*]$. If we take the best-first strategy in line 7, we can also take $[f^*] = [\underline{f},\ \tilde{f}]$ with $\underline{f} = \min\{\underline{F}([y]),\ [y] \in \mathcal{L}\}$.

algorithm 5 A new Interval Branch & Bound Algorithm for Global Optimization

1: **Input:** $[\mathbf{x}]^\mathbf{0}$ starting box, ϵ tolerance for the stopping criterion,
2: F inclusion function for f
3: **Output:** $\tilde{f}$, approximation for f^* and $\mathcal{S}$, list of boxes covering S^*
4: $\tilde{f} = \overline{F}(mid([x]^0))$
5: initialize work list $\mathcal{L} = ([x]^0, \underline{F}([x]^0))$, $\mathcal{S} = \varnothing$
6: **while** $\mathcal{L}$ is not empty **do**
7: **choose** a pair $P = ([x], \underline{F}([x]))$ from $\mathcal{L}$
8: **if** stopping criterion holds **then**
9: insert P into $\mathcal{S}$ and **goto 6**
10: **end if**
11: $\tilde{f} = \min\{\tilde{f}, \overline{F}(mid([x]))\}$ {update the minimum}
12: split the box $[x]$ into $([x]_1, \ldots, [x]_n)$ { sub-dive [x]}
13: compute $F([x]_i)$ for $i = 2 \ldots n$ and store $([x]_i, \underline{F}([x]_i))$ in $\mathcal{L}$
14: $[x]_1$.monotonicity_test()
15: $[x]_1$.one_newton_test() using $[f^*] = [-\underline{F}([x]^0),\ \tilde{f}]$ { the new test }
16: $[x]_1$.concavity_test()
17: $[x]_1$.newton_step()
18: perform **cut_off** test on $\mathcal{L}$
19: insert what remains from $[x]_1$ into $\mathcal{L}$
20: **end while**

We placed the new test *one_newton_test* just after the monotonicity test to reuse the value $[g_i]$ of the computed enclosure of the gradient. The only overhead due to the new test is the computation of $G(m([x]))$, nevertheless the experimental results we are presenting in the next section show that it is in general worth applying this test.

2.6.4 Experimental results and remarks

The new discarding test presented above is similar, to some extends, to slopes techniques proposed by Ratz, see [44]. It is also similar to the branch-and-prune

method for global optimization presented in [50]. The first difference between the new test, the one-newton-test, and these others techniques is that, the later are for univariate functions. Moreover, the basic idea behind the pruning technique of Ratz is to incorporate to Algorithm 1 a new discarding test in lieu of the "powerful" monotonicity test when the objective function is not smooth. While, the technique presented here can merely be used as a new discarding test.

Tables 2.3 and 2.4 show the performance of different versions of Algorithm 1 on some standard test problems. The first version of the algorithm is without the new test and the second version included this test. In Table 2.4 we suppose that the global minimum is known, this is the case for some global optimization problems. We used a heap to manage subproblems. We inserted a box $[x]$ in the solution list when $w(F([x]) < \epsilon_F = 10^{-6}$, see Section 2.4.1. These experimental results were carried out on a Pentium-IV machine (2.8 Ghz, 1 Gbyte) under the Linux operating system. The time unit is the second.
These experimental results show that it is definitely worth considering this test as a new acceleration device. In many cases when applying the new test, the performance achieved by the interval global optimization algorithm is better than the performance obtained by the algorithm without this test. Only in few cases we observed that, with this test, the algorithm is slower but never significantly. We have an overall improvement of about 25% when the minimum is not known and of about 40% when the minimum is known.

This test could also be applied for the *set inversion problem* and in *constraints propagation*, but in this case the algorithm obtained could be less efficient because one would have to explicitly compute the gradient of the function, which is not the case in global optimization where one already has the value of the gradient via the monotonicity test. But if one uses centered forms to compute the inclusion function then the value of the gradient would be available as well.

To apply this test we choose the directions cyclicly w.r.t the sub-boxes hierarchy. The test could be more efficient if one would be able to choose a *good* direction based on some indicators functions.

Problems	Old (Without the test)					New (With the test)					Improvement
	Iter	fEval	gEval	hEval	Time	Iter	fEval	gEval	hEval	Time	
R4	540	1578	1095	463	0.15	466	1725	953	285	**0.13**	**(13.33%)**
GP	3895	13517	8010	3506	2.42	3715	14114	7644	3281	**1.89**	**(21.9%)**
SHCB	600	1508	1093	355	0.32	515	1563	932	269	**0.29**	**(9.1 %)**
MS	358	934	713	209	**0.05**	358	1066	714	210	0.06	**(-1.0 %)**
SW	3100	15288	11472	4658	2.03	1819	10332	6876	2766	**1.21**	**(40.39%)**
JS	301	1105	787	280	0.24	283	1152	786	276	**0.19**	**(20.83%)**
Gro	593	1664	1224	412	1.56	563	1522	826	187	**1.54**	**(1.28%)**
Gr0	53	1700	1301	430	**2.97**	591	2103	1304	433	3.05	**(-2.2 %)**
HM3	49264	147311	71834	21007	22.61	28959	151534	65727	22609	**16.75**	**(25.91%)**
HM3*	425536	1180367	849451	339084	225.95	182602	7366440	357627	138851	**109.41**	**(55.55%)**
HM2	28868	85901	45526	14514	10.49	16534	65582	33220	127727	**7.04**	**(32.88%)**
HM2*	42710	123443	85623	38220	16.73	22312	89254	44734	16906	**9.77**	**(40.23%)**
L3	32596	99100	54678	17769	13.49	17273	71032	37607	13539	**8.78**	**(34.94%)**
L3*	119679	358976	252243	112142	58.41	69724	295804	151013	58652	**37.45**	**(36.22%)**
L11	20050	59549	38158	17563	28.39	16209	65416	32046	15268	**24.89**	**(12.32%)**
KOW	222678	868440	589588	235988	388,71	220758	918781	584696	233850	**385.1**	**(0.92%)**
WK	28588	122689	73682	32996	24.43	21054	92408	50229	23288	**17.06**	**(30.16%)**
Siirola	2604624	7149394	6107712	2384268	99271.8	1250107	4726341	2677634	1022173	**52274.3**	**(47.34 %)**
											(25.79%)

Tab. 2.3: Performance of the versions of the algorithm on some standard test problems, the minimum is unknown

Problems	Old (Without the test)					New (With the test)					Improvement
	Iter	fEval	fEval	hEval	Time	Iter	fEval	gEval	hEval	Time	
R4	540	1578	1094	463	0.14	423	1620	868	254	**0.11**	**(21.12%)**
GP	3895	13517	8010	3506	2.17	3715	14114	7644	3281	**2.00**	**(7.5 %)**
SHCB	600	1508	1093	355	0.29	487	1472	879	244	**0.26**	**10.3 %)**
MS	358	934	713	209	**0.05**	358	1066	714	210	0.06	**(-1.0 %)**
SW	717	3063	1988	817	0.33	427	2078	1209	494	**0.19**	**(42.43%)**
JS	301	1105	787	280	0.19	273	1119	768	270	**0.19**	**(0.0%)**
Gr0	53	1700	1301	430	**2.97**	591	2103	1304	433	3.05	**(-2.2 %)**
HM3	49264	147311	71834	21007	22.61	17594	67531	33378	12395	**9.74**	**(56.92%)**
HM3*	425536	1180367	849451	339084	227.95	153840	572079	298225	102989	**86.5**	**(60.26%)**
HM2	28868	85901	45526	14514	10.41	13234	51628	26477	9521	**5.59**	**(46.30%)**
L3	32596	99100	54613	17704	13.8	17065	70202	37108	13358	**8.59**	**(37.759%)**
L3*	119679	358976	252243	112142	58.42	68570	290433	148521	57408	**34.56**	**(38.42%)**
L11	20050	59549	35770	15491	23.83	2212	9435	4378	2085	**3.33**	**(86.15%)**
KOW	222672	867900	588671	235716	**383.82**	220722	917983	583820	233576	384.85	**(-0.26%)**
WK	72332	287801	226215	73386	62.09	42834	203533	118318	45643	**36.55**	**(41.11%)**
Siirola	1605624	3244694	2207452	1284264	77269.4	850517	1725344	1275435	700134	**24537.8**	**(68.34 %)**
											(39.17%)

Tab. 2.4: Performance of the versions of the algorithm on some standard test problems, the minimum is known

Chapter 3

A New Parallel approach

This chapter is concerned with the parallelization of the branch-and-bound algorithm for global optimization presented in the previous chapter. We begin here with a presentation of parallel computing issues. A few words will be said about issues such as memory organization, flow control, interconnection network, etc. The second part of this chapter is about issues in the design of approaches for the parallelization of the interval global optimization algorithm. We first present existing methods, then we present a new approach based on the distribution of the work of the root processor.

3.1 Parallel computing issues

In this part we explain basic issues of parallel computing. We make comparisons of parallel computers and describe the hardware characteristics of parallel computers. For more details one can see [37, 7, 36, 38].

Definition 4 (see [36]) *A high performance parallel computer is a computer that can solve large problems in a much shorter time than a single desktop computer. These computers are characterized by fast CPUs, large memory, a high speed interconnect, and high speed input/output. They are able to speed up computations; both by making the sequential components run faster and by doing more operations in parallel.*

One distinguishes two type of processes, namely, sequential processes and parallel processes.

Definition 5 *Sequential processes are those that occur in a strict order, where it is not possible to do the next step until the current one is completed. Parallel processes are those in which many events happen simultaneously.*

There are many examples of such sequential and parallel processes in our daily live. Let's look at the parallelism in computer programs. It is well known now that almost all computer programs can lend themselves to parallelism. One should only determine which part of the program can be executed simultaneously. Given a program that takes a time t to be executed on a single processor machine, the ideal in parallel computing is to have the program executed in time t/p on a p processors machine or using p such machines.
To execute a program in parallel, one checks in general whether its program has independent parts. Part P_1 is independent of part P_2 if the execution of P_1 does not affect P_2 and vice versa. One also tries to determine if the data needed by the program can be processed simultaneously. Dealing with these two issues is what one calls *decomposition*. Two types of parallelism therefore exist: *Data parallelism* and *Task parallelism*.

Data parallelism In data parallelism the same code segment runs concurrently on each processor, but each processor is assigned its own part of the data to work on.

Task parallelism Instead of the same operations being performed on different parts of the data, each process performs different operations.

3.1.1 Parallelism in computers

Parallelism in computers intervenes at many levels. Parallelism is exploited at the operating system level, arithmetic units level, and memory and disk management level to enhance computation.

Performance Measures

There are numerous ways of measuring performance of a parallel computer or a parallel program. Each performance measure is briefly described.

MFLOPS (Millions of Floating Point Operations Per Second) measures how quickly a computer can perform floating-point operations such as add, subtract, multiply, and divide. A gigaFLOPS (GFLOPS) is equal to one billion (10^9) floating-point operations per second. A teraFLOPS (TFLOPS) is equal to one trillion (10^{12}) floating-point operations per second.

Peak Performance is the top speed at which the computer can operate. It is a theoretical upper limit on the computer's performance. The speed is usually measured in MFlops, GFlops or TFlops.

Sustained Performance is the highest consistently achieved speed for a given application. It is a more realistic measure of computer performance.

Cost Performance is used to determine if the computer is cost effective.

Speedup measures the benefit of parallelism. It shows how a program scales as it is executed using more processors, compared to the performance on one processor. Ideal speedup happens when the performance gain is linearly proportional to the number of processors used. Let t_s denote the time needed by a sequential algorithm to solve a problem and t_p the time needed by a parallel algorithm to solve the same problem with p processors. The speedup of this algorithm with p processors is

$$S(p) = \frac{t_s}{t_p}.$$

Accordingly, one defines the *efficiency* of this algorithm as

$$E(p) = \frac{S}{p} = \frac{t_s}{p \cdot t_p}.$$

If $E(p) \geq c > 0$ for all p, then one has a linear speedup. $E(p) = 1$ is the optimal speed-up. If $E(p) > 1$, then one has a *superlinear* speedup. A superlinear speedup is unusual and indicates that the sequential algorithm can be improved. For efficient serial algorithms, a superlinear speedup is typically due to the fact that intensive memory requirements are handled more efficiently in parallel.

Benchmarks are used to rate the performance of parallel computers and parallel programs. A well known benchmark that is used to compare parallel computers is the Linpack benchmark. Based on the Linpack results, the Top 500 Supercomputer list is produced biannually. This list is maintained by the University of Tennessee and the University of Mannheim. See http://www.top500.org/.

Load balancing is all about keeping processors busy by efficiently distributing the workload. In particular, an optimal load balancing method will have the following general characteristics:

- computation is equally distributed across all processors
- throughput of all applications in the system is maximized
- response time of single requests is optimized
- task scheduling achieves the quickest execution of all tasks

3.1.2 Comparison of Parallel Computers

Parallel computers can be classified according to:

- number and type of processors
- memory organization
- flow of control
- interconnection networks

Processors

One can distinguish three situations:

1. Computers with a small number of extremely powerful (vectors) processors. typically some tens of processors. The cooling of these computers often requires very sophisticated and expensive equipment, making these computers very expensive for computing centers.

2. Computers with a large number of less powerful processors. Often named a **M**assively **P**arallel **P**rocessor (MPP). They typically have thousands of processors. The processors are usually proprietary and air-cooled. Because of the large number of processors, the distance between the farthest processors can be quite large requiring a sophisticated internal network that allows distant processors to communicate with each other quickly. These computers are suitable for applications with a high degree of concurrency.

3. Computers that are medium scale in between the two extremes. Such medium scale computers typically have hundreds of processors. The processor chips are usually not proprietary; rather they are commodity processors like the Pentium IV. These are general-purpose computers that perform well on a wide range of applications. The most common example of this class are Linux Clusters, like ALICEnext, the 1024 AMD opteron processor cluster at the university of wuppertal. For more details see http://www.alicenext.uni-wuppertal.de/.

Memory Organization

One finds three types of memory organization on parallel computers:

1. distributed memory

2. shared memory

3. distributed shared memory

Distributed memory In distributed memory computers, the total memory is partitioned into memory that is private to each processor. There is a **N**on-**U**niform **M**emory **A**ccess time (NUMA), which is proportional to the distance between the two communicating processors. On NUMA computers, data is accessed the quickest from a processor's own private memory, while data from the most distant processor takes the longest to access. Some examples of distributed

memory parallel computers are the `Cray T3E`, the `IBM SP`, and workstation clusters.

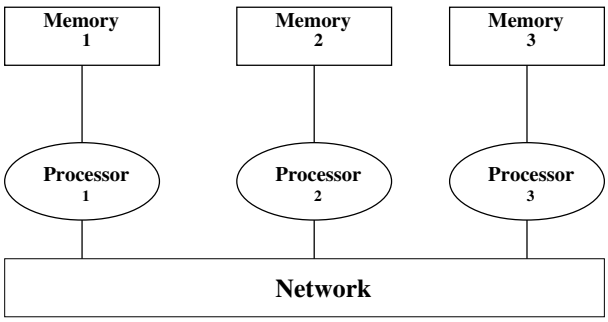

Fig. 3.1: Distributed memory computer

Distributed memory computers use message passing such as MPI to communicate between processors.
One advantage of distributed memory computers is that they are easy to scale. As the demand for resources grows, computer centers can easily add more memory and processors. The drawback is that programming of distributed memory computers can be quite complicated and that the network may become a bottleneck.

Shared memory: In shared memory computers, all processors have access to a single pool of centralized memory with a uniform address space. Any processor can address any memory location at the same speed so there is **U**niform **M**emory **A**ccess time (UMA). Processors communicate with each other through the shared memory.

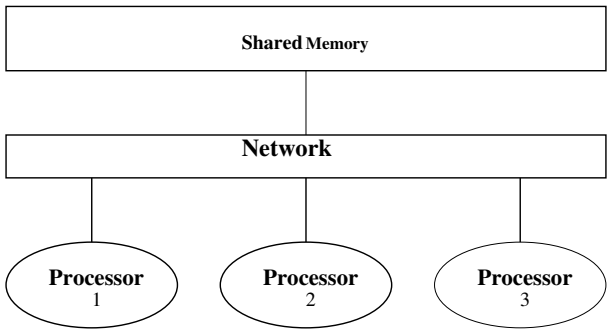

Fig. 3.2: Shared memory computer

The advantages and disadvantages of shared memory machines are roughly the

opposite of distributed memory computers. They are easier to program because their programming resembles that of single processor machines, but they don't scale like their distributed memory counterparts.

Distributed shared memory: In **D**istributed **S**hared **M**emory (DSM) computers, a cluster or partition of processors has access to a common shared memory. It accesses the memory of a different processor cluster in a **NUMA** fashion. Memory is physically distributed but logically shared. Attention to data locality again is important.
Distributed shared memory computers combine the best features of both distributed memory computers and shared memory computers. That is, DSM computers have both the scalability of distributed memory computers and the ease of programming of shared memory computers. Some examples of DSM computers are the `SGI Origin2000` and the `HP V-Class` computers.

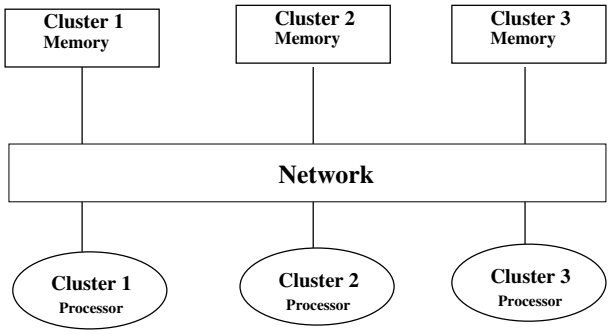

Fig. 3.3: Distributed shared memory computer

Flow control

According to the control of flow, one has three types of parallel computers:

1. **S**ingle **I**nstruction **M**ultiple **D**ata (**SIMD**)
2. **M**ultiple **I**nstruction **M**ultiple **D**ata (**MIMD**)
3. **S**ingle **P**rogram **M**ultiple **D**ata (**SPMD**)

SIMD Computers: **SIMD** stands for **S**ingle **I**nstruction **M**ultiple **D**ata. There is a single instruction stream, so that each processor follows the same set of instructions. But there are multiple data streams, with different data elements being allocated to each processor. SIMD computers may have distributed memory with typically thousands of simple processors, and the processors run

in lock step. SIMD computers, popular in the 1980s, are useful for fine grain data parallel applications, such as neural networks. Some examples of SIMD computers were the Thinking Machines `CM-2` computer and the computers from the `MassPar` company.

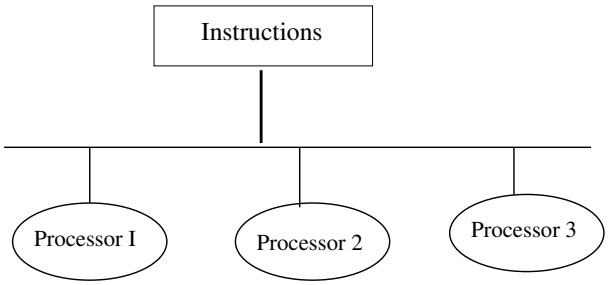

Fig. 3.4: SIMD diagram

MIMD Computers: **MIMD** stands for **M**ultiple **I**nstruction **M**ultiple **D**ata. There are multiple instruction streams with separate code segments distributed among the processors. MIMD is actually a superset of SIMD, so that the processors can run the same instruction stream or different instruction streams. In addition, there are multiple data streams; different data elements are allocated to each processor. MIMD computers can have either distributed memory or shared memory. While the processors on SIMD computers run in lock step, the processors on MIMD computers run independently of each other. MIMD computers can be used for either data parallel or task parallel applications. Some examples of MIMD computers are the `SGI Origin2000` computer and the `HP V-Class` computer.

SPMD Computers: **SPMD** stands for **S**ingle **P**rogram **M**ultiple **D**ata. SPMD is a special case of MIMD. SPMD execution happens when a MIMD computer is programmed to have the same set of instructions per processor. With SPMD computers, while the processors are running the same code segment, each processor can run that code segment asynchronously. Unlike SIMD, the synchronous execution of instructions is relaxed. An example is the execution of an if statement on a SPMD computer. Because each processor computes with its own partition of the data elements, it may evaluate the right hand side of the if statement differently from another processor. One processor may take a certain branch of the if statement, and another processor may take a different branch of the same if statement. Hence, even though each processor has the same set of instructions, those instructions may be evaluated in a different order

from one processor to the next.

Memory	SIMD distributed memory	MIDM distributed or shared memory
Code segment	same per processor	same or different
Processors run	in lock step	asynchronously
Data elements	different per processor	different per processor
Applications	data parallel	data parallel or task parallel

Tab. 3.1: Summary of SIMD versus MIMD

Interconnection Networks

The interconnection network is made up of the wires cables and interfaces that define how the multiple processors of a parallel computer are connected to each other and to the memory units. The time required to transfer data is dependent upon the specific type of the interconnection network. This transfer time is called the communication time.
Possible network topologies (geometric arrangements of the computer network connections) are:

- Bus
- Cross-bar Switch
- Hypercube
- Tree
- Mesh or Torus

The aspects of network issues are: **cost**, **scalability**, **reliability**, **suitable applications**, **data rate**, **diameter**, **degree**.

Definition 6 (Degree) *how many communicating wires are coming out of each processor. A large degree is a benefit because it allows for multiple paths in the graph defining the interconnection network.*

Definition 7 (Diameter) *This is the distance between the two processors that are farthest apart. A small diameter corresponds to low latency.*

Bus Network: Bus topology is the original coaxial cable-based **L**ocal **A**rea **N**etwork (LAN) topology in which the medium forms a single bus to which all stations are attached.
The benefits of a bus based network are that it is simple to construct. It is also a mature technology that is well known and reliable. Since bus based networks are so common, the cost is also very low.
The negative aspects to a bus based network are that it has a limited data transmission rate. In addition to this the most significant problem is that it is also not scalable in terms of performance.
When too many processors try to talk to each other over a bus based network, the communication slows down and slows down the performance of the program. An example of a computer with this type of network is the `SGI Power` Challenge. The Power Challenge only scaled to 18 processors.

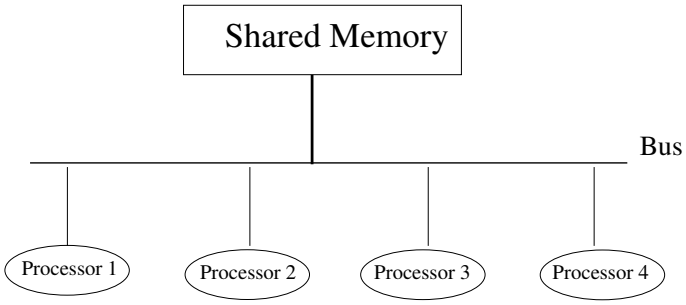

Fig. 3.5: Bus network diagram

Cross-bar Switch Network: A cross-bar switch is a network that works through a switching mechanism to access shared memory. One benefit is that it scales better than the bus network but it costs significantly more.
The telephone system uses this type of network. An example of a computer with this type of network is the HP V-Class. Below is a diagram of a cross-bar switch network which shows the processors talking through the switch boxes to store or retrieve data in memory. There are multiple paths for a processor to communicate with a certain memory. The switches determine the optimal route to take.

Hypercube Network: In a hypercube network, the processors are connected as if they were corners of a multidimensional cube. Each node in an N dimensional cube is directly connected to $N-1$ other nodes. The fact that the number of directly connected, "nearest neighbor", nodes increases with the total size of

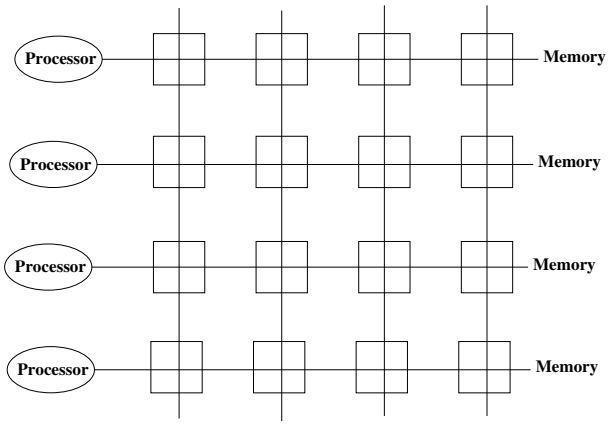

Fig. 3.6: Cross-bar switch network diagram

the network is also highly desirable for a parallel computer. The degree of a hypercube network is $\log n$ and the diameter is also $\log n$, where n is the number of processors. Examples of computers with this type of network are the `CM-2`, `CUBE-2`, and the `Intel ipso860`.

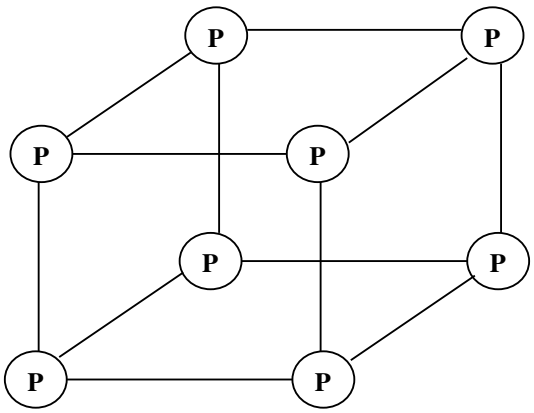

Fig. 3.7: Hypercube network diagram

Tree Network: The processors are the bottom nodes of a tree. For a processor to retrieve data, it must go up in the network and then go back down. This is useful for decision making applications that can be mapped as trees. The degree of a tree network is 3. The diameter of the network is $2\log(n+1)-2$ where n is the number of processors.

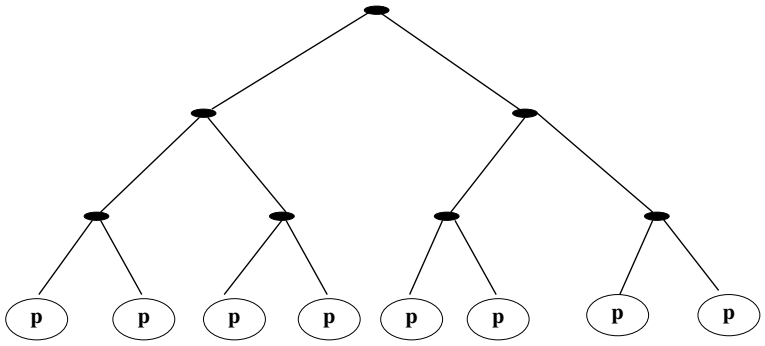

Fig. 3.8: tree network diagram

Mesh or Torus: In a mesh network, the nodes are arranged in a k dimensional lattice of width w, giving a total of w^k nodes. A torus network is obtained from a mesh by wraparound connections between nodes at the borders of the mesh. Schematically we have Figure 3.1.2 and 3.1.2.

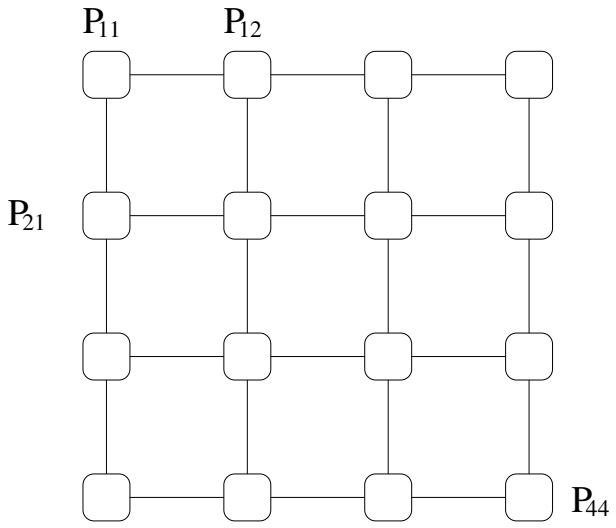

Fig. 3.9: 2D mesh network diagram

The ALICEnext case

We want to give here the characteristics of the machine on which the computation in the parallel case were carried out.
ALICEnext was in summer 2004 the most powerful parallel computer at a German university and is now (July 2005) the number 167 in the top500 list.

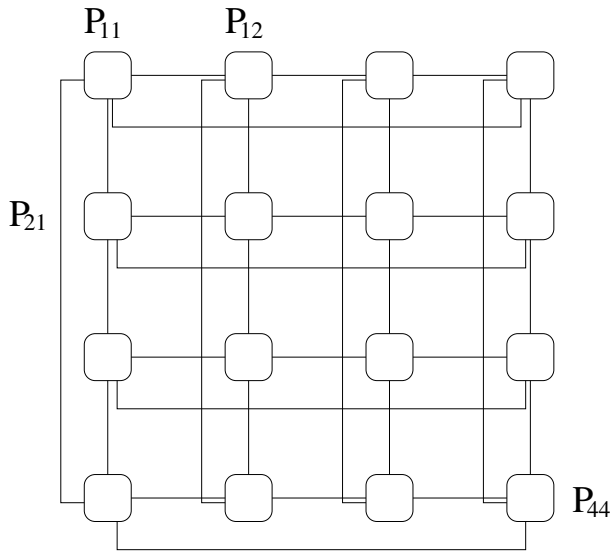

Fig. 3.10: 2D torus network diagram

ALiCEnext consists of 1024 AMD-Opteron processors distributed on 512 blades. On each of these blades one has, two AMD Opteron 1.8 GHz processors, two 250 GB hard discs, two 1024 MByte RAM, 6 x Gigabit-Ethernet connections. The 512 blades are mounted in 11 towers, 48 blades per tower in 4 rows of 12 blades. The network has two layers. One layer arranges 32 groups of 16 blades as 2d-toruses. The other is a hierarchical switch-based network with 64-pat switches arranged in 4 levels.

3.2 Parallel global optimization issues

The algorithm we want to parallelize belongs to the branch-and-bound category. As was explained in the first chapter, branch-and-bound means that from the original problem, subproblems, of not necessarily the same size, but of the same type, are generated. Whereas for many linear algebra problems, the amount of work at each node can be estimated at the beginning, it is very difficult (almost impossible) to do the same for branch-and-bound algorithms. This is due to their irregular and unpredictable computational behavior. It is then clear that static load balancing which very often is efficient for many linear algebra problems will tend to be inefficient for branch-and-bound algorithms. We therefore have to dynamically load work on processors. Dynamic load balancing means that *during* the execution process, tasks should be evenly scheduled among the involved processors. Even if, due to the branch-and-bound principle, interval

global optimization lends itself to parallelism, load balancing is by no way a straightforward task as it seems to appear. In fact the parallelization of this algorithm is subject to many compromises.
It turns out that the main issue in the parallelization of the interval global optimization algorithm is the *dynamic* load balancing. Two types of load balancing exist for this task, namely, *quantitative load balancing* and *qualitative load balancing*. Quantitative load balancing, in the interval global optimization context, is responsible of having the same number (or at least almost the same number) of boxes on each processor. Qualitative load balancing is responsible of having processors working on most promising boxes. The latter means that the p processors involved in the parallel process should be working on the p most promising boxes.
The following issues are to be considered in the design of parallel approaches for the global optimization algorithm.

- Keep all processors busy, of course doing useful jobs.
- Provide processors as fast as possible with the newly updated lower bound $\tilde{f}$ of the minimum. Since $\tilde{f}$ is used to discard boxes in the cut-off test, the aim is to avoid handling boxes that would not have been handled in the sequential algorithm.

Basically all approaches found in the literature to solve the global optimization deal with these issues, the way they do it makes the originality of each of them. Two methods exist to manage the distribution of subproblems. The first is to store problems in a central processor, the other is to distribute these subproblems on all available processors.

3.2.1 Management of subproblems

Centralized list

Here a central processor keeps the list of boxes and provides other processors (workers) with these boxes when they need them. When a processor has finished processing a box it sends what remains from this box back to the central processor. The advantage here is that it is likely that processors will be working on most promising boxes if the central processor uses best-first as its selection strategy. Therefore, qualitative load balancing is achieved. The other advantage might be the easy implementation of methods based on this idea. The obvious disadvantage is that the total memory available is limited to the memory of the

central processor. This is a serious handicap for interval global optimization which requires a lot of memory to keep subproblems, particularly for difficult problems. Moreover it could be very expensive to migrate boxes after each iteration; the central processor is likely to become a bottleneck very soon.

Distributed list

Here each processors manages its local list as in the serial case. The advantages and disadvantages of the distributed list are roughly the opposite of the centralized list. Processors are unlikely to be working on most promising boxes. However, this disadvantage is negligible compared to the advantage of having small local list and of using the whole memory available. We use a distributed list in the design of our method. We next present some existing methods, for more details see [20] and references therein.

3.2.2 Existing approaches for global optimization

The Approach of Dixon and Jha (1993)

The parallelization in [52] takes place on a transputer net with $p = 13$ T800 Transputers. They are arranged in a tree where each node has three children. The transputer in the root of the tree manages the list of boxes. If there are more than p boxes in the list then each processor handles some box. Otherwise the next box of the list is subdivided orthogonally to one direction into p parts which are distributed between processors. This method was tested on five test functions. The speedup was disappointing. Using 13 processors, the speedup was between 2.83 and 8.75. In the majority of test functions it was less than 4.

The Approach of Henriksen and Madsen (1992)

This approach was implemented on a net of T800 transputers, see [25] For parallelization, a master-slave principle was used. The master manages a central list $\mathcal{L}$ of boxes and the upper bound $\tilde{f}$. It sends boxes from the list to slaves, who in turn send back the result to the master. A result is a pair consisting of a box and a lower bound, and the updated value for $\tilde{f}$. The master sends the best (smallest) $\tilde{f}$ to the rest of the processors (see Figure 3.11). The starting box is subdivided into $p-1$ parts at the beginning, where p is the number of processors used.

The program was tested on 1, 4, 8, 16 and 32 processors. When passing from 16 to 32 processors in most cases the speedup increases only a little or even decreases much. This decrease was observed even earlier for the majority of test

functions. One of the main reasons of the decrease of speedup is the overhead for communication.
To reduce communication, in [25] the depth-first-strategy was used instead of the best-first strategy. This has the advantage that the slaves need not to get a box from the central processor in each iteration. Instead, after bisection, they keep one box for further handling and send only the second box back to the master (if it is not discarded). The slave must request a box from the master only if both boxes are discarded or fulfill the stopping criterion. In [25] the upper bound $\tilde{f}$ was initialized to the global minimum f^*. In this case, all selection strategies become equivalent in the sense that they handle exactly the same boxes, though in different order. The number of boxes to handle is minimized. The speedup for the depth-first-method was almost always better than with the best-first-method (also initialized with $\tilde{f} = f^*$), sometimes quite significantly. But even using the depth-first-strategy, the speedup does not increase any more or increases only a little bit for larger value of p. For some test problems the speedup for 32 processors was below 16. For the others the speedup on 32 processors was between 19 and 28.

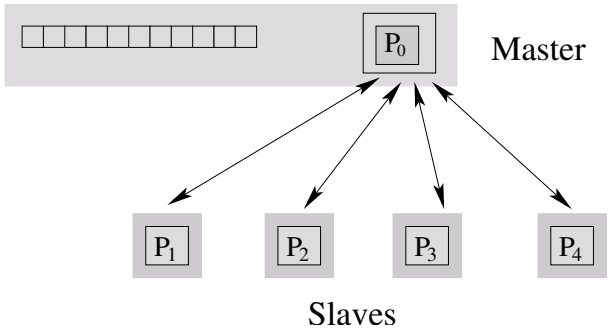

Fig. 3.11: The master-slave model: The master manages the central sorted list and the upper bound $\tilde{f}$. The boxes are sent to the slaves for handling, and the results are received back.

The Approach of Eriksson (1991)

This method, see [13], which was implemented on a iPSC/2 Hypercube, arranges all processors in a pool. The processors are logically located in a ring. There is an orientation in the ring, so that there is a next one for every processor. On every processor there are two processes running: a worker and a scheduler. The worker is responsible for handling boxes and for the distribution of the upper bound $\tilde{f}$. The scheduler is responsible for load balancing of boxes left

to be handled. On each processor the scheduler manages its own list of boxes. When the worker has found a better upper bound $\tilde{f}$, it sends this value to all processors immediately. This becomes possible by using an asynchronous receive-command available on the iPSC/2 that indicates readiness of another worker. A signal sent to a worker is immediately received and its readiness is reset. Several approaches on how the scheduler does the load balancing were tested. Balancing with respect to the quantity of boxes was implemented using the receive-initiated-scheduling. When the worker on a processor has handled a box and the list of the scheduler is empty, the scheduler sends a request-message to the next processor in the ring. If this processor has no boxes to give, it retransmits the request further. The first scheduler whose list is not empty sends the box to the scheduler of the process which initiated the request. The box will not travel in the ring, but is sent to the corresponding processor directly.
Balancing with respect to the quality of boxes was implemented as sender-initiated-scheduling. In this way one tries to achieve that boxes with small lower bounds are handled as soon as possible. More precisely, one tries to handle the p most promising boxes analogously to the serial method. Since we have local lists of boxes, it is difficult to guarantee that really the p most promising boxes are handled. The following scheme was developed in [12]: If the number of boxes on a certain processor is greater than the given limit (it was set to 5), then the processor sends its first box to a randomly selected processor. One modification of this approach is to use a dynamic limit instead of a static one. If the box inserted is in the head (lists are sorted), then the limit is decremented, otherwise it is incremented. In this manner good boxes are sent early. On the other hand the processors with less promising boxes send boxes only rarely, since their limit is incremented. Numerical results presented in [12] show that a method which uses receive-initiated-scheduling combined with dynamic send-initiated-scheduling is efficient. Through the usage of the sender-initiated- scheduling the total number of boxes handled for a given problem is reduced. The speedup for the three considered problems were 9.71, 19.58 and 11.97 on 16 processors and 15.04, 28.26 and 30.88 on 32 processors, respectively. So superlinear speedup was achieved for one problem (on 16 processors). The reason for superlinear speedup was not explained in [13].

The Approach of Moore, Hansen and Leclerc (1992)

As opposed to the methods considered so far the parallelization of Moore, Hansen and Leclerc in [33] is based on the serial method that uses the oldest-

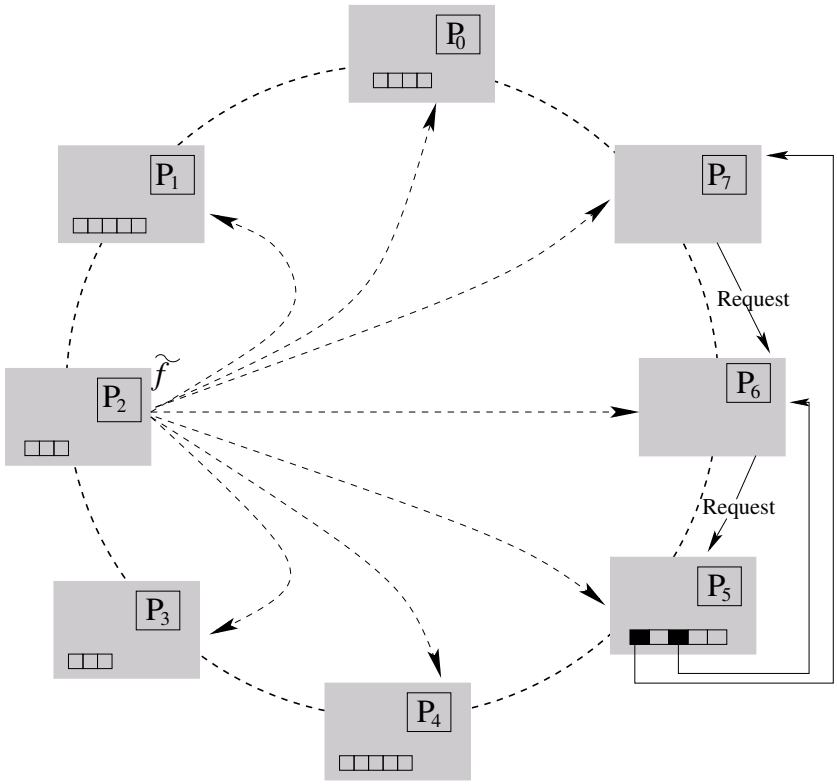

Fig. 3.12: Communication structure by Eriksson

first-strategy for box selection. To accelerate the method along with the midpoint test and the monotonicity test also the nonconvexity test and the interval Newton method were used. The parallel method was implemented on a workstation cluster of 250 Sparc-Stations SLC. Like in the approach of Eriksson every processor manages its own list. If processor P_i has no more boxes to handle, then it sends a request to a randomly selected processor P_j. If processor P_j has boxes in the list, then it sends half of its boxes (but no more than a limit set a priori) to the processor which initiated the request. Otherwise P_i sends a request to $P_{(j+1) \bmod p}$, $P_{(j+2) \bmod p}$ and so on (see Figure 3.13). Running this parallel method on the parameterized problem MHL (see [1]), superlinear speedup was achieved. A maximum speedup of 170 on 32 processors was achieved.

The Approach of Berner (1995)

Berner's approach described in [2] was implemented on a CM-5, a MIMD computer with 32 processors. The parallel method is based on the serial method that uses the monotonicity test, the nonconvexity test and the interval New-

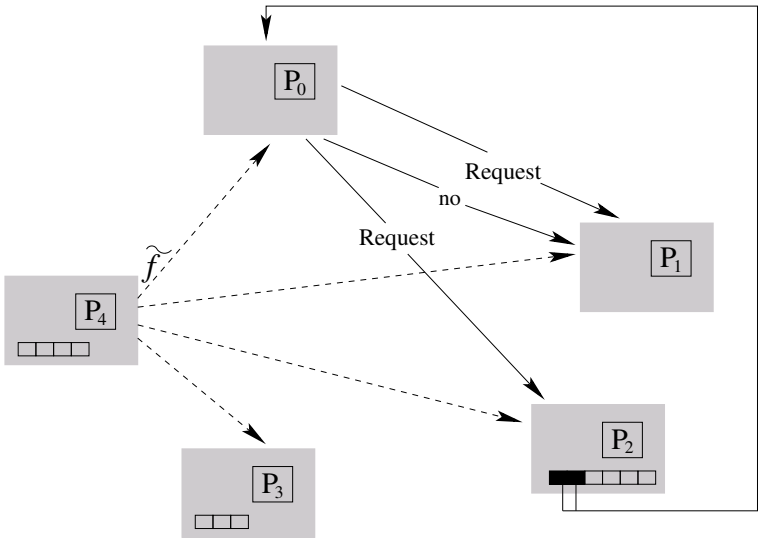

Fig. 3.13: Communication structure in the parallelization by Moore, Hansen and Leclerc

ton method as accelerating devices. As a box selection strategy the best-first-strategy is chosen. In this approach there is one centralized mediator and many workers (see Figure 3.14). Each worker manages its own list of boxes whose length is controlled by the centralized mediator. The centralized mediator waits for requests of idle processors to send them new boxes. Moreover, it keeps a limit max, that is changed dynamically. This limit is used to make sure that the centralized mediator does not run out of boxes, but also that not too many boxes are stored in its list. Processors which keep more than max boxes in their lists send some of them to the centralized mediator (see Figure 3.14). The boxes to be sent to the centralized mediator are selected neither randomly nor from the tail. Every second box (at most max-send boxes are sent) is selected from the list. Each processor sends the best upper bound to all workers and the centralized mediator. An advantage of this parallel approach compared to the master-slave model used in Henriksen and Madsen, see Section 3.2.2, is that there is less work for the centralized mediator than for the master. So it will not become a bottleneck if the number of processors used is not too large. Moreover, the whole memory including that of the workers is used. Compared to the approach of Eriksson and the one of Moore, [13], Hansen and Leclerc, [33], there is no need to request several processors to get boxes if a processor becomes idle. Instead, it is the centralized mediator that directly responds to each request. The method was run on 4, 8, 16 and 32 processors. For some test

problems slight superlinear speedup was achieved.

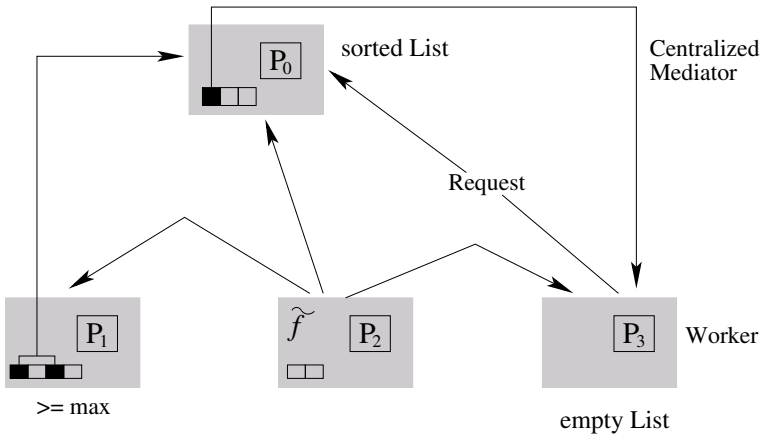

Fig. 3.14: Communication structure by the parallelization by Berner

The Approach of Wiethoff (1997)

In [53] Wiethoff presented his distributed parallel method. It was implemented on an IBM RS/6000 SP. Up to 96 processors were used. For the list management he used the best-first-strategy. His parallel method was based on a serial method with accelerating devices like the monotonicity test, the nonconvexity test and the interval Newton method. Boxes were subdivided into 4 parts. Processors were located in a pool logically arranged in a ring. All processors had their own lists of boxes. Each processor communicates only with its 4 neighbors (two nearest and two next to nearest). On every processor two processes run. One for load balancing and exchange of the best upper bound. The other is for handling local boxes. The newly found better upper bound is sent only to neighbors and from there propagated further. This method was run on 4, 8, 16, 32, 64, 96 processors. A total of 18 problems were tested. For a few problems superlinear speedup was achieved. On 8 processors for 2 problems, on 16 processors for 3 problems, on 32 and more processors only for one problem. The method has no communication bottleneck at all. But the larger the number of processors the lower the efficiency, since information is then distributed very slowly.

The Approach of Ibraev (2001)

Described in [20], this method uses the advantage of the best first strategy and the centralized mediator. In this model there is always one leader and many workers (see Figure 3.15). The leader is determined dynamically as the processor holding the smallest best upper bound. The leader has boxes for handling.

Therefore idle processors send requests to the leader. When a processor obtains a better upper bound, it sends a challenge to the leader but not to other processors. The leader determines the smallest of the best upper bounds, if it receives several of them, and decides who is the next leader. It sends the new best upper bound together with the information on change of the leadership in one message to all other processors. If the leader runs out boxes, i.e. boxes in its list have been discarded or put into the solution list, it chooses any non-idle processor as the next leader. In the case that there is no non-idle processors left, it sends a termination signal to all the processors and the method ends. One advantage of this method is that idle processors receive boxes from the processors having the best upper bound, therefore it is likely that processors will be working on promising boxes. For problems where there are many local minima near the global minimum, this method could be inefficient since there are many promising boxes and there is no need to "challenge" processors. In fact in this case, every non-idle processor has "good enough" boxes that could be sent to every idle processors. Recall that one of the characteristics of many difficult problems is that there are many local minima near the global minimum. This method was implemented on a cluster of SUN machines. In [20] it was shown that for many problems superlinear speedup was achieved.

In Section 3.2.4 we will make a comparison with the new strategy we designed.

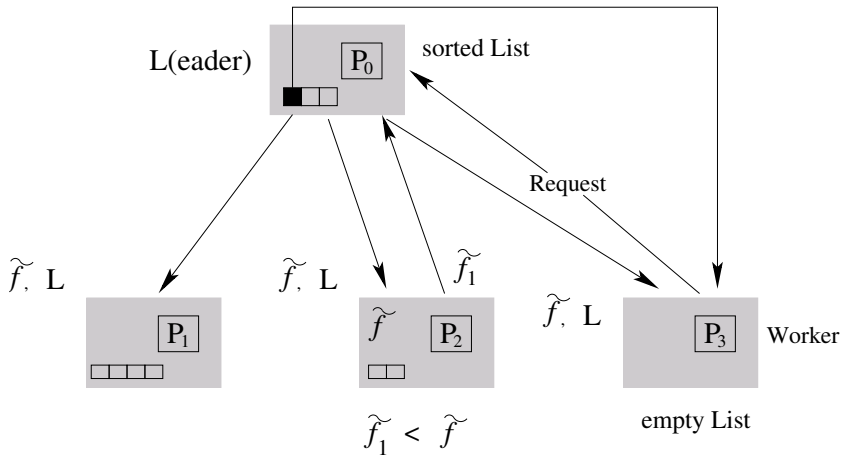

Fig. 3.15: Communication model in Ibraev's challenge leadership

3.2.3 A new approach: Distributed Management

When communication in the parallel process becomes intensive, almost all methods presented have the common disadvantage that a root (master, leader, mediator ...) processor is likely to become a bottleneck. This phenomenon is difficult to avoid when the number of processors grows, since the root processor has to listen to all possible communications with other processors. We propose in the sequel an approach to avoid this situation. The main idea behind this method is to alleviate the work of the root by allowing other processors (non idle ones) to serve requests coming from idle processors.

Description

The aim of this method is to combine the qualitative and the quantitative load balancing while trying to avoid the bottleneck effect. Each processor has its own local list. The root processor (with rank 0) maintains a list containing the number of boxes other processors have in their list. This is for a quantitative load balancing purpose. The main difference with other strategies is the behavior of the root processor when processors run out of boxes, that is when processors are idle.

When processors become idle. In this case the root processor does not automatically send boxes to these idle processors, instead, it determines which non idle processors process should provide these processors with boxes. It proceeds as following. It creates two groups of processors. One group of *idle* processors and one group of *non idle* processors. It sorts, in decreasing order, with respect to the length of the list, the group of non-idle processors. This means that, one has $|\mathcal{L}(p_i)| \geq |\mathcal{L}(p_{i+1})|$, for two consecutive processors in the group of non-idle processors, where $|\mathcal{L}(p_i)|$ denotes the length of the list of the processor of rank i. The root then establishes a correspondence between these two groups of processors. The first processor in the group of non-idle processor sends a number N of boxes to the first idle processor. The second processor in the group of non-idle processors should provide the second idle processor with boxes and so on. The number of idle processors could be different to the number of non-idle processors, in this case the distribution starts over cyclically with the first non idle processor. That is, suppose we have 3 non idle processors (p_1, p_2, p_3) and 5 idle processors (p_4, p_5, p_6, p_7, p_8) in this order in their respective groups; in this case P_1 sends boxes to p_4 and p_7; p_2 sends boxes to p_5 and p_8, and p_3 sends boxes to p_6. Figure 3.16 illustrates this mechanism. In this case there are more busy processors than idle ones. Figure 3.17 shows the case where there

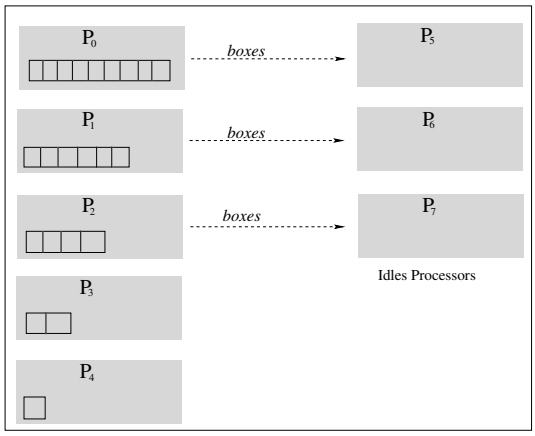

Fig. 3.16: Distribution of boxes in the new method, 5 non-idle processors, 3 idle.

are more idle processors than busy ones.

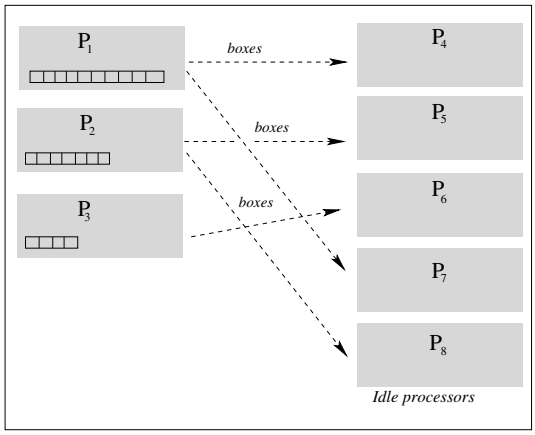

Fig. 3.17: Distribution of boxes in the new method, 3 non-idle processors, 5 idle.

Figure 3.18 shows all possible communications among processors. Here *Info* means that the root processor is exchanging some information with a worker. *Info* contains information such as the number of boxes this processor should send to a particular idle processor. *Request* means that a processor ran out of boxes and wants some boxes. *New bound* means that a processor obtained a smaller value for $\tilde{f}$. *Length* means that a processor is sending the length of its list to the root processor.

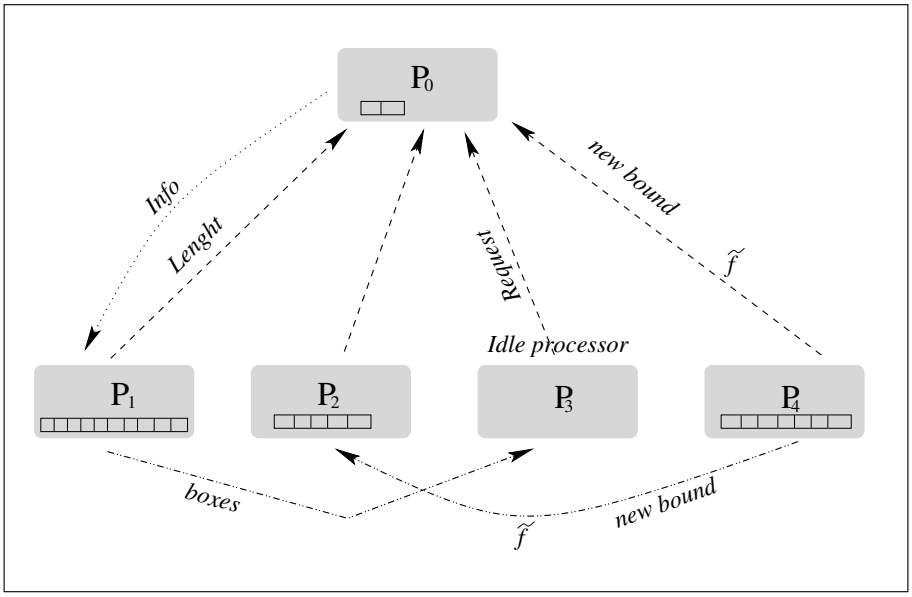

Fig. 3.18: Illustration of communications in the new method.

To discuss our new method we present both advantages and disadvantages it has over other methods.

Disadvantages

- The root processor must know the length of the list of all other processors. This is not easy to achieve and could produce some overhead. Each processor must send the length of its list to the root. If they do this too often, the root processor will become a bottleneck very soon. If they do it too rarely, the root might be working with old, non-updated values. The pace at which processors should send their length to the root should be determined dynamically, we describe later how we do it.

- Boxes are not directly sent to idle processors since the root must determine which processor should provide idle processors with boxes. So the idle times of processors are somewhat increased.

These disadvantages are negligible compared to the advantages we present next.

Advantages

- In general the processor with the largest list is likely to have the most promising boxes. Therefore, qualitative load balancing is implicitly achieved.

Thus with this strategy, both quantitative and qualitative load balancing are likely to be achieved simultaneously.

- Idle processors are provided with boxes simultaneously. In other strategies, the root must first send boxes to the first $i-1$ idle processors before the ith idle processor sees itself provided with boxes. Figure 3.19 illustrates the distribution of boxes in these strategies. Here, if the number of processors increases, the root node will soon become a bottleneck. In high dimension where the time to send a box could be significant, the new method is more efficient.

- There is no need to move boxes among processors. An idle processor receives boxes directly from its provider.

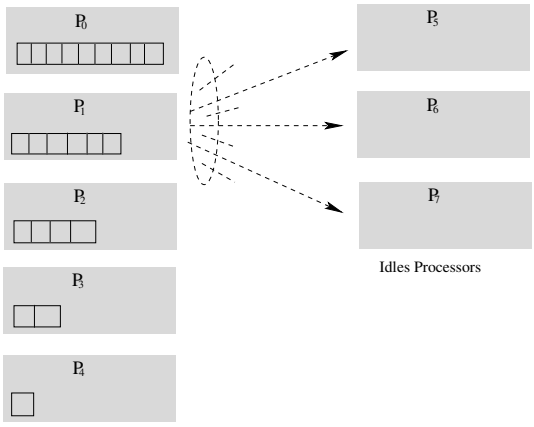

Fig. 3.19: Distribution of boxes in the old methods, three idle processors and five non idle processors, P_1 is likely to become a bottleneck.

Broadcasting the new upper bound of f^*

In general, when a processor wants to share information with all other processors, the maximal time a processor must wait to have such an information is linear in the number of processors involved. The aim of this part is to show how one can reduce this time. One typical case for such a need is when a processor obtains a better upper bound $\tilde{f}$ for the minimum. In this very case, optimized broadcast routines available within MPI are not usable here since they are collective routines which suppose that a broadcast must be posted on all processors. But the processor which found a better upper bound is the only one which knows that, since memory is distributed.

Suppose that P_b obtained such a $\tilde{f}$, suppose further that there are n processors involved. In the standard approach processor P_b would send this $\tilde{f}$ to the other $p-1$ processors one after the other. That is, P_b will send $\tilde{f}$ to $P_i,\ i \in \{1,\ldots,n\},\quad i \neq b$ in the order $1 \ldots n$. Let t be the time needed to send a message containing $\tilde{f}$. In this case, processor P_n will receive the message containing $\tilde{f}$ after a time $T = (n-1)t$. Asymptotically, this means that the maximal time a processor should wait to obtain a message is linear in the number of processors.
We propose to minimize this time. To do this, we arrange processors in a g-ary tree. In this virtual topology, processors communicate only with their neighbors. The nodes of the tree represent processors. If a processor P_b wants to send $\tilde{f}$ to other processors, it sends it to the root processor P_0, the root processor forwards this value to its sons, which forward this value to their own sons, and so on. This process will continue until leaf processors are reached. Schematically we have Figure 3.20 with 13 processors. With the same number

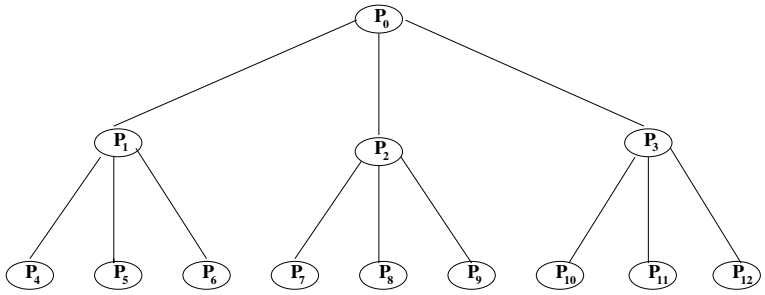

Fig. 3.20: A tree as a virtual topology to broadcast messages

of processors but sending messages to processors one after each other, one has Figure 3.21. Suppose that processor P_0 wants to send a message to P_{12}. Using

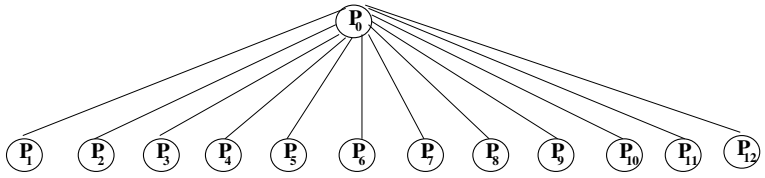

Fig. 3.21: Sending messages to processors one after others

the virtual topology with a 3-ary tree like the one on Figure 3.20, we see that one needs a time $T = 1 + 2 \times 3t = 7t$ to have the message on P_{12}. Sending this message with the *one-after-the-other* strategy, one would need $12t$.

Now, more generally, let's do a simple analysis in the case where one has n processors arranged in a g-ary tree where the leaf level is not necessarily full. In this case, the maximal time to wait will occur for leaf processors. Suppose again that t is the time required to send a message, then the total time to wait for each processor is less or equal to

$$T(g) = (1 + gh) \cdot t \tag{3.1}$$

where

$$h = \lfloor \log_g(n) \rfloor.$$

In (3.1), we add 1 to gh because the processor which found the best upper bound $\tilde{f}$ will first send it to the root processor.
For a fixed number n of processors the value of T depends on g. Let's find the value of g for which $T(g)$ is minimal. Replacing h in (3.1) by $\log_g(n)$, we have the following estimation

$$T(g) = (1 + g \log_g(n)) \cdot t.$$

Suppose for a while that $g \in \mathbb{R}$ and $g > 1$, i.e. $g \in (1, +\infty)$ so that we can write

$$T'(g) = \frac{(\ln(g) - 1)}{(\ln(g))^2} \cdot \log(n) \cdot t.$$

With a short calculation we see that $T'(g) = 0$ for $g = e$ and $T''(e) > 0$, meaning that $g = e$ is the minimum of T. Moreover, T is decreasing before e and increasing after e.
Now, since $g \in \mathbb{N}$ and $g \geq 2$, it follows that T is minimum for $g = 2$ or $g = 3$. Returning to (3.1), we see that $T(3) > T(2)$ so that $T(g)$ is minimal for a binary tree.

This technique could also be used, with the communication pattern reversed, to send the length of the lists of the processors to the root. This has the advantage not only to obtain the length of the list of processors faster, but also to avoid the bottleneck at the root processor. A possible drawback here could be the latency. For slow networks interconnection where the latency is high, the overhead due to this technique could be noticeable since some processors have to receive and forward messages. Still, in the case where this strategy is used to send best upper bounds $\tilde{f}$, there is an advantage due to the fact that a processor could update the value of the global minimum it received. Therefore, after a broadcasting, processors could receive different values of $\tilde{f}$. The efficiency of such a technique is noticeable when the number of processors involved is very large.

Used communication routines

We describe our algorithm using pseudo code. Processors communicate by sending messages. We give here the meaning of each subroutine and variable we are using in the pseudo code of functions we are presenting later.

size: indicates the number of processors involved in the parallel process.

myrank: indicates the rank of the current processor (the calling processor)

root: indicates the root processor and is set equal to 0 for conveniences of implementation.

Msg: This is a parameter for all communication routines. We assume, for the sake of simplicity, in the pseudo codes, that it contains all necessary fields. For example, when sending or receiving boxes, *Msg.Boxes* contains the list of boxes to be sent or to be received. We also suppose that for a box B, $B.min$ is the lower bound of the function over B, i.e. $B.min = \underline{F}(B)$. We actually send pairs $(B, \underline{F}(B))$, not only boxes.

To specify the type of messages a processor will receive or send, message passing tools use tags. Here are the tags used in our pseudo codes.

tag_box: This tag indicates that a processor is sending or receiving boxes.

tag_empty: This tag is used when a processor ran out of boxes and wants to notify that to the root.

tag_length: This tag is used by a worker to send the length of its list to the root. The root also uses this tag to receive this length.

tag_not_enough: This tag is used when a processor does not have enough boxes to send to idle processors.

tag_new_minimum: This tag indicates that the message to be received contained a new value for the global minimum $\tilde{f}$.

tag_provide: This tag is used by the root to ask a non idle processor to send boxes to an idle one.

tag_finish: This tag is used to mention the end.

tag_sol: This tag is used to send and receive solution boxes.

We have limited to three the number of communication routines to make the presentation of pseudo codes easier. We explain what they mean.

Probe(Msg): With this routine, we mean that the calling processor is checking if there are pending messages. **Msg** is an output argument in this case containing all information needed to determine what type of message it is. **Msg.flag** is a boolean indicating whether there is a pending message. If **Msg.flag = true**, then there is a pending message, otherwise there is no pending message.

Send(Msg, tag, destination): The processor which calls this routine is sending a message contained in **Msg** with the tag **tag** to the processor whose rank is **destination** .

Receive(Msg, tag, source): Receives the pending message with tag **tag** from the processor with rank **source** in the variable **Msg**.

Broadcast(Msg): With this routine, the variable **Msg** is broadcasted to all processors. This is used by a processor when it found a better value for the minimum.

Description of the algorithm

Algorithm 6 begins by setting the root equal to the processor with rank 0, this could have been any other processor. This part of the algorithm is executed by all processors. After that, the algorithm is subdivided into two parts; one part executed by the root and the other executed by workers. The root processor reads the input and distributes parts of the starting box to workers. The root subdivides the starting box so that each processor knows which part it should work on. We describe next the first functions called by the root, namely, Initial_Phase() and Distribut_Input(). The aim of this procedure is to produce a number of boxes a least equal to the number of processors. This procedure uses Algorithm 2 presented in Section 2.4.6 with $l = \lceil \log_2(size) \rceil$ as argument. Thus, a least *size* subboxes are generated. Some authors, see [2], prefer to run some few steps of the sequential algorithm to produce subboxes. Merely subdividing the starting box into the number of processors available has the disadvantage that some processors could receive subboxes for which there is nothing to do, because they already fulfilled the stopping criteria, consequently there will be an immediate need of balancing. The second idea - running few steps of the

algorithm 6 The Parallel Global Optimization Algorithm

Input: $[\mathbf{x}]^{\mathbf{0}}$ starting box, ϵ the tolerance, f the function, P the number of processors

Output: f^* the minimum and S the set of global minimizers

```
1: root = 0                                   {the root is the processor with rank 0}
2: if (myrank == root) then
3:    Initial_Phase()
4:    Distribut_Input()
5:    while (!FINISH) do
6:       handle_Root_Request()
7:       handle_Root_boxes()                          {when necessary}
8:    end while
9:    Terminate()
10: else                                                    {for workers}
11:   Receive_Input()
12:   while (!FINISH) do
13:      handle_Worker_Request()
14:      handle_Root_boxes()
15:   end while
16:   Terminate()
17: end if
```

algorithm 7 Initial_Phase()

Input: $[\mathbf{x}]^{\mathbf{0}}$ starting box *size* the number of processors **Output:** S the set of subboxes

$r = \lceil \log_2(size) \rceil$
S = Bisect($[x]^0$, r);

sequential algorithm - has the advantage that a fairly good load balancing could be achieved at the beginning due to the fact that processors will receive boxes for which one is almost sure that the stopping criteria would not immediately be satisfied. But such a starting phase would depend on the problem under consideration and could be inefficient when the number of processors involved is large. With many processors this initial phase could spend time uselessly since a number of boxes equal to the number of processors should be produced. Depending on the problem under consideration the time needed to perform this initial phase could be very significant compared to time the whole algorithm

will take. In this procedure the root sends boxes obtained from the initial phase

algorithm 8 Distribut_Input()

Input: S set of subboxes from Initial_Phase()

```
while S not empty do
  for i = 0 to length of S do                          {for workers}
    B = S.pop()                                        {remove B from S}
    Send(B, tag_box, P_((i+1) mod size))
  end for
end while
```

to workers. If *size* is not a power of 2 then the number of boxes does not equal the number of processors. In this case the root continues the distribution cyclically with the processor with rank 1. Therefore, processors may receive different numbers of boxes. This does not have a significant impact on the performance of the algorithm.

The procedure corresponding to *Distribute_Input* is the worker's procedure *Receive_Input*. With this procedure each processor receives its part of the starting

algorithm 9 Receive_Input()

```
Receive(B, tag_My_Part, root)        {receive the initial box from the root}
WorkList = B;     {set the work list to B and continue with handle_box()}
```

box. Now processors have boxes in their list, they can handle them, this is done in *Handle_Worker_Boxes*.

This procedure actually does what the sequential algorithm does. The algorithm has a global variable *NumOfIter* which indicates how many times the sequential algorithm *Handle_Box()* should be called. The idea is that processors should not be querying for requests neither too often nor too rarely. Without the *while* (line 3) loop in the algorithm presented above, processors would be probing for new messages after each iteration. The consequence would have been some additional overheads. *NumOfIter* should not be too large, otherwise another processor could wait too long before its request is handled.

As soon as a new value of the minimum is found, it is broadcasted to the other processors, line 8. Workers also send the length of their list to the root. To avoid sending this information too often, the worker checks whether the length of its list has changed significantly since the last send. It sends this information when $|MyLength - OldLength| > size$, where $OldLength$ indicates the length

algorithm 10 Handle_Worker_Boxes()

```
1: f̃old = f̃;
2: Count = NumOfIter;
3: while (Count ≠ 0) do
4:    B = WorkList.pop();
5:    Handle_Box(B);                          {handle B as in the serial case}
6:    Count = Count - 1;
7: end while
8: if ( f̃ ≠ f̃old) then
9:    broadcast(f̃);                           {broadcast the new value of the minimum}
10: end if
11: if | MyLength − OldLength |> size then
12:    Send(MyLength, tag_length, root);       {send length of list to the root}
13:    OldLength = MyLength;
14: end if
15: if (WorkList.empty()) then
16:    Send(char, tag_empty, root);
17: end if
```

of the processor during the last send and $MyLength$ the current length of its list.
If the processor runs out of boxes, i.e. WorkList is empty, then it sends a request to the root with the tag *tag_empty*.
Now we present the corresponding procedure for the root.

The root processor handles boxes only if it does not have too many requests to serve. It handles boxes when the number of requests is less than a threshold. We set this threshold equal to the number of processors. If the root processor obtains a better value for the minimum, it broadcasts this value too. If it runs out of boxes, it asks for boxes from the worker having the largest list. This worker is the worker whose rank is in ListOfRequest[0]. This *ListOfRequest* variable is used by the root to evenly balance work among processors. We give more details about that in the sequel.
We now describe the *Handle_Request()* procedure, where the essential work of the parallel process is done. We first present the *Handle_Request()* procedure on the worker's side. The worker first checks if it has pending messages. If this is the case, it determines the nature of the message.
If the message has a tag *tag_new_minimum* then it means that a processor has

algorithm 11 Handle_Root_Boxes()

```
f̃old = f̃;
Count = NumOfIter;
threshold = size;
if (NumOfRequest < threshold ) then
  while ( Count ≠ 0) do
    B = WorkList.pop();
    handle_box(B)                          {handle B as in the serial case}
  end while
end if
if (f̃ ≠ f̃old) then
  Broadcast(f̃);                {broadcast the new value of the minimum}
end if
if (WorkList.empty()) then
  Send(char, Tag_Empty, Request[0]);
end if
```

algorithm 12 Handle_Worker_Request()

```
f̃old = f̃;
repeat
  Probe(Msg)
  if (Msg.flag == true) then
    if (Msg.tag == tag_new_minimum) then
      Update_Minimum();                    {updating the minimum}
    else if (Msg.tag == tag_box) then
      Receive_Box(Msg);                          {receiving boxes}
    else if (Msg.tag == tag_provide) then
      Serve_Box(Msg);          {sending boxes to idle processors}
    else if (Msg.tag == tag_not_enough) then
      Serve_not_enough();
    else if (Msg.tag == tag_finish) then
      Serve_Finish(Msg);
    end if
  end if
until Msg.flag == false
```

found a new approximation for the global minimum and wants to share it with other processors. The processor calls the procedure *Update_Minimum* below to update the value of $\tilde{f}$. If the message is a message with tag *tag_box* then the

algorithm 13 Update_Minimum()

Input: Msg

```
Receive(Msg);
f̃new = Msg.f̃;
f̃ = min (f̃, f̃new)
```

processor calls the procedure *Receive_Box* below.

algorithm 14 Receive_Box()

Input: Msg

```
Receive(Msg);
List = Msg.Boxes
while (List is not empty) do
  B = List.pop();
  if (B.min ≤ f̃) then
    WorkList.push(B);          {inserting boxes in the WorkList}
  end if
end while
```

With this procedure, idle processors receive boxes and transfer them in their working list. But before inserting boxes, they perform the cut-off test.
If the message has a tag *tag_provide* then it means that the root is asking a worker to send boxes to an idle processor. In this case, the variable *Msg* contains the ranks of processors to which boxes should be sent. This variable also contains the number of boxes that should be sent. Below is the procedure called in this case.
In this procedure, the calling processor sends boxes from its list to idle processors. If the processor can not provide all idle processors with boxes, it sends a message with tag *tag_not_enough* to processors which did not received boxes. When those processors will receive a message with tag *tag_not_enough*, they will resend a message with tag *tag_empty* to the root, signaling that they are still idle.
If the message has a tag *tag_not_enough* then *Serve_not_Enough* is called.

algorithm 15 Serve_Box()

Input: Msg

```
Receive(Msg);
List = Msg.List                    {list containing the ranks of processors}
NumBox = Msg.NumBox                {number of boxes to send}
while List is not empty and WorkList.size() > 1 do
  rank = List.pop();
  while WorkList.size() ≥ 1 and NumBox ≥ 1 do
    List1 = List1 + WorkList_pop(NumBox);        {pop boxes to send}
  end while
  Send(List1, tag_box, rank);                    {sending boxes ...}
end while
while List is not empty do
  rank = List.pop();
  Send(char, tag_not_enough, rank);
end while
```

algorithm 16 Serve_not_Enough()

Input: Msg

```
Receive(Msg);
Send(char, tag_empty, root);
```

Here the worker sends a new message with tag *tag_empty* to the root since it did not receive boxes.
If the message has a tag *tag_finish* it means that the root has sent a termination message, in this case *Serve_Finish()* is called.

algorithm 17 Serve_Finish()

Input: Msg

```
Receive(Msg);
FINISH = true;                                   {it is the end}
```

Having finished describing the procedure *handle_Worker_Request*, we now present the procedure *Handle_Root_Request*. To balance the work among processors, the root has variables containing the statuses of other processors. It has a variable *ListOfRequest* which is a vector of size $size-1$. This variable contains the length of the list of other processors. When the root receives a new length it updates

the length of the corresponding processor calling the procedure *serve_length*. The root can also receive a message with a tag *tag_box*, in this case it receives boxes by calling the procedure *Receive_Box* presented earlier. The root can receive a new value for $\tilde{f}$, in which case it calls *Update_Minimum* presented above. When the root receives a message with tag *tag_empty*, it sets the entry of the corresponding processor in the variable *ListOfRequest* to 0, meaning that this processor is idle.
When the root has finished receiving messages, it calls the procedure *balance*, see below.

algorithm 18 Handle_Root_Request()

```
f̃old = f̃;
repeat
  Probe(Msg)                                  {looking for new messages}
  if (Msg.flag == true) then
    if (Msg.tag == tag_new_minimum) then
      Update_Minimum();                       {updating the minimum}
    else if (Msg.tag == tag_Box) then
      Receive_Box(Msg);                       {receiving boxes}
    else if (Msg.tag == tag_Length) then
      Serve_Length(Msg);                      {receiving length of workers}
    else if Msg.tag == tag_empty) then
      Serve_empty();
    end if
  end if
until Msg.flag == false
balance();                          {creates the two groups of processors ...}
```

We give the description of the procedure in the algorithm above. Many of these procedure are very simple. For the sake of clarity and explanation we list them separately.

algorithm 19 Serve_Length()

Input: Msg, Tag, Source

```
Receive(Msg, Tag, Source);
rank = Msg.rank;
ListOfRequest[rank] = Msg.Length;
```

algorithm 20 Serve_Empty()

Input: Msg, Tag, Source

```
Receive(Msg, Tag, Source);
rank = Msg.rank;
ListOfRequest[rank] = 0;
```

For these two procedures we suppose that Msg contains the rank and the length of the list of the processor that sent the message.
The aim of *balance* is to determine which processors should send boxes to idle processors. The procedure begins by creating a list of non idle and a list of idle processors, lines 3 and 6. It tests whether the size of the list of idle processors equals the number of processors minus 1 (the number of workers), if this is the case then, all workers are idle. If the root has no box, it means that the algorithm terminates. The root then sends a message with tag *tag_finish* to workers. If the root has boxes, it sends them to idle processors. Now, if there are some non idle processors, then the root sorts the list of these non idle processors, in a decreasing order, with respect to the length of their list; this information is contained in *ListOfRequest*. The first processor in the list of non idle processors (the processor with the largest list) is asked by the root to send *NumToSend* boxes to one idle processor. *NumToSend* is equal to the sum of boxes on all processors divided by the number of processors. The idea is to have almost the same number of boxes on all processors. When a processor has been asked to send boxes, the root assumes it has done so and the length of its list is updated, see line 30. The length of its list is set to the number of elements it had before minus the number of elements it has been asked to send, and the process starts again until there is no idle processor.
The next procedure to describe is *terminate*. This procedure is called at the end by all processors. In this function workers send boxes in their *SolutionList* to the root and exit. The root in this function receives solution boxes from workers.

algorithm 21 balance()

Input: Msg

```
1: SUM = 0;
2: for ( i = 1 to size − 1 ) do
3:    if (ListOfRequest[i] == 0) then
4:       ListOfIdle.push(i);                          {setting a processor as idle}
5:    else
6:       ListOfNonIdle.push(i);                       { setting a non idle processor}
7:    end if
8:    SUM = SUM + ListOfRequest[i];
9: end for
10: if (ListOfIdle.size() == size - 1) then          {they are all idle, it might be the end }
11:    if ( MyLength == 0) then
12:       FINISH = true;
13:       for ( i = 1 to size − 1) do
14:          Send(FINISH, tag_finish, i);                {Terminating ...}
15:       end for
16:    else
17:       while (ListOfIdle is not empty) do
18:          B = WorkList.pop();
19:          Send(B, tag_box, ListOfIdle[i]);          {sending boxes to workers }
20:       end while
21:    end if
22: else
23:    NumToSend = SUM / size;                        {number of boxes to send}
24:    Msg.NumToSend = NumToSend;
25:    while ( ListOfIdle is not empty) do
26:       Sort(ListOfNonIdle);                        {sorting the list of non idle }
27:       rank = ListOfIdle.pop();
28:       Msg.rank = rank;
29:       Send(Msg, tag_provide, ListOfIdle[1]);  {sending the rank of idle processor to non
          idle one}
30:       ListOfRequest[ListOfNonIdle[1]] = ListOfNonIdle[1] - NumToSend;
31:    end while
32: end if
```

algorithm 22 Terminate()

```
if (myrank == root) then
  probe(Msg);
  while (Msg.flag) do
    probe(Msg.flag);                          {receiving solution boxes ...}
    Receive(Msg, tag_sol, Msg.rank);
    B = Msg.Box;
    SolutionList.push(B);                     {pushing the solution boxes ...}
  end while
else
  Send(SolutionList, tag_sol, root);
end if
```

Superlinear speedup?

Superlinear speedup achieved by algorithms in linear algebra or numerical computing almost always indicates that the serial algorithm used to measure the speedup is not efficient. In general superlinear speedup can be expected when the serial algorithm requires a lot of memory. In the global optimization context superlinear speedup can be due to the fact that the whole amount of work in the parallel case is less than the amount of work in the serial case.
Superlinear speedup of a parallel global optimization method can be due to two factors. Firstly, superlinear speedup can be achieved because of the memory required by the problem under consideration. In fact, if the subproblems generated in the sequential case don't fit in the memory available then the algorithm will begin to swap to the disk and this will result in a very slow sequential program. In the parallel case, since the whole available memory is used, this phenomenon could not be observed, or at least will be delayed, provided the subproblems are evenly distributed. This is the first reason for superlinear speedup to occur, but this is not really specific to parallel global optimization. In the global optimization context, superlinear speedup can also be due to the fact that a good approximation for the global minimum in the parallel process is obtained faster than in the serial case. This is due to the fact that, in the parallel case many boxes are considered simultaneously. If a good approximation for the global minimum is obtained, it will be broadcasted to other processors and these can perform the cut-off test, discarding boxes that will have been considered by the sequential algorithm. Consequently, the number of boxes considered in the parallel process could be less than the number of boxes considered by the

sequential algorithm and a superlinear speedup could follow.

Experimental results and remarks

In this section we comment on experimental results obtained running the parallel algorithm on some standard test problems.
In Table 3.2 we recorded the time needed by the algorithm on some standard test problems as a function of the number of processors. The second column ($p = 1$) corresponds to the serial algorithm. The unit of time in the table is second. The test environment is ALICEnext see section 3.1.2. Processors communicate using Message Passing Interface (**MPI**) routines. On Figure 3.22, 3.23, 3.24, 3.25 we plotted the speedup versus the number of processors, up to 64.

	Number of processors						
Problems	1	2	4	8	16	32	64
R4	2.54	1.31	0.72	0.41	0.30	0.31	0.34
SHCB	0.68	0.36	0.24	0.25	0.29	0.28	0.30
MS	0.13	0.8	0.7	0.6	0.5	0.9	0.10
JS	2.16	1.09	0.53	0.29	0.14	0.10	0.6
SW	13.56	6.45	3.19	1.89	0.98	0.51	0.28
Gro	16.54	8.23	4.09	2.01	1.03	0.6	0.31
GP	4.67	2.28	1.04	0.66	0.40	0.22	0.27
HM3	2050.23	1093.32	536.6	274.06	142.06	76.3	40.02
HM2	1207.56	742.3	372.6	141.6	67.86	35.85	20.5
L3	1235.22	606.5	298.3	152.2	77.99	40.8	23.1
L11	3010.22	1589.53	756.3	373.6	153.21	91.3	55.01
Siirola	30512.77	13235.62	6862.21	3285.4	1506.54	835.6	471.5
KOW	385.1	196.6	91.6	47.65	22.65	13.25	7.25
WK	389.35	185.54	85.6	40.15	21.6	12.64	7.23
INF1	826.94	394.6	198.31	94.74	42.65	23.09	13.13

Tab. 3.2: Performance of the parallel algorithm on some standard test problems

Figure 3.22 resumes the speedup achieved on some simple test problems. These problems require few iterations, typically less than 500. The time required is generally less than 5 seconds. This explains the bad speedup obtained. In fact, these problems not really require parallelization. Such problems, with few

iterations, could require parallelization if the optimization problem is a part of a whole process as it is the case in robotic. For the sake of clarity, we limited the presentation only to 4 test problems. The behavior is the same for all problems in this category. Figure 3.23 shows the speedup of the algorithm

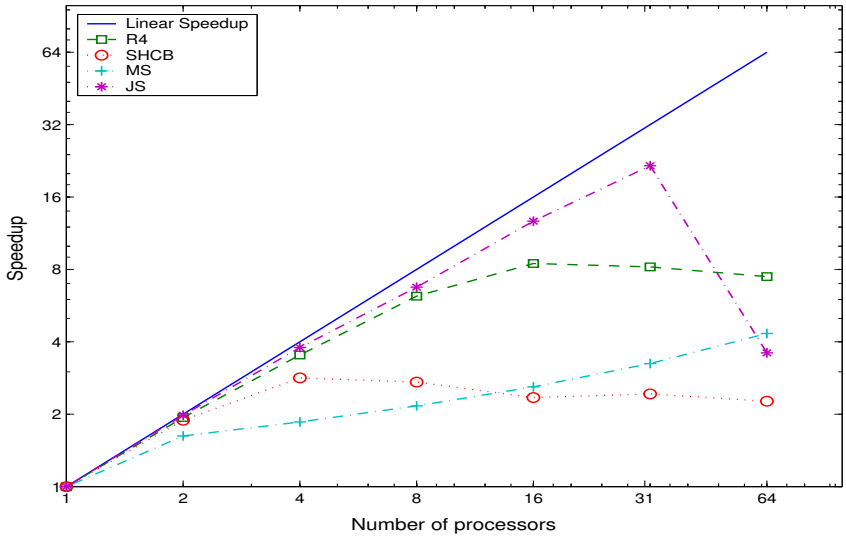

Fig. 3.22: Speedup of the parallel algorithm on some simple problems

on some medium test problems. The speedup is a bit better than in the case of simple problems. For a small number of processors the speedup is almost linear. One observes an overall improvement of the speedup over the simple test problems.
In Figures 3.24 and 3.25 we have the performance of the algorithm on difficult test problems. The speedup is always almost linear. Superlinear speedup is even sometimes attained. The most important positive aspect here is that the speedup decreases only a little bit when the number of processors grows. The serial algorithm requires a lot of iterations and produces many boxes on these problems. In the parallel case, these boxes are distributed among all processors. It turns, that processors will be, for a relative long time, busy. With many processors (32, 64), it likely that many processors become idle simultaneously. The *Distributed Management* has the advantage that these idle processors receive boxes almost simultaneously. Therefore, the time to wait for boxes is optimal (minimal). Moreover, it is likely that they receive promising boxes.

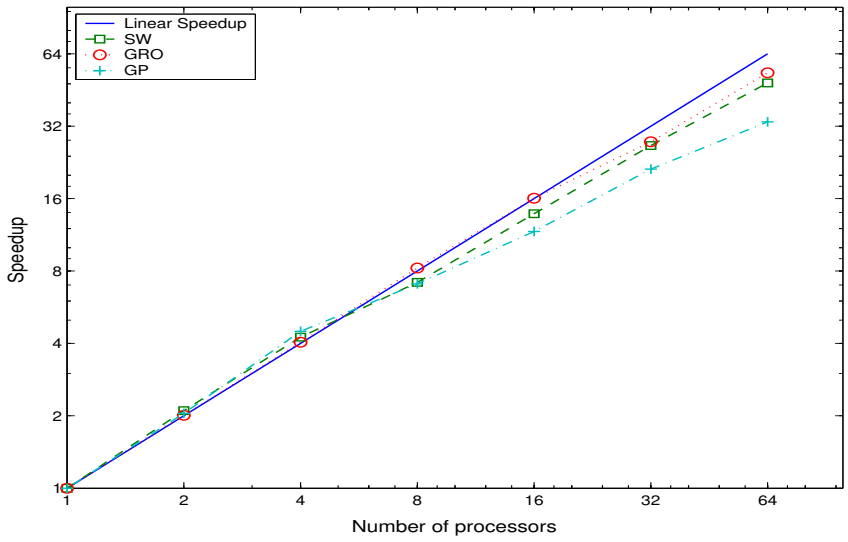

Fig. 3.23: Speedup of the parallel algorithm on some medium problems

3.2.4 Benchmarking

Difficulties with benchmark

Difficulties faced with the practical comparison of parallel approaches are inherent to the heuristical behavior of accelerating devices used in the interval global optimization algorithm. As a matter of fact, settings such as which accelerating devices to use and when to do so, do have an influence on the performance of the serial algorithm. So does the interval library in use too.
Now, considering the fact that for difficult problems, the interval global optimization algorithm requires a lot of memory to store subproblems, it turns out that parallel algorithms, no matter which approach is used, will achieve a good performance, provided the memory of all processors is used. This is because one has more memory and these subproblems are likely to fit in the whole available memory. It follows that the data structures used by the serial algorithm plays a key role in the analyze of the parallel approaches.
In Section 3.2.2 we gave a description of existing parallel approaches. As far as we know, none of these approaches implements an appropriate data structure such as heap to store subproblems. It is then likely that, in the serial algorithm used to calculate the speedup, most of the time is spent to handle the list. In many cases, for difficult problems, the algorithm would even swaps to disk, resulting in a very inefficient serial algorithm. This could explain the embarrassingly high speedup achieved by these parallel approaches. Figure 3.26

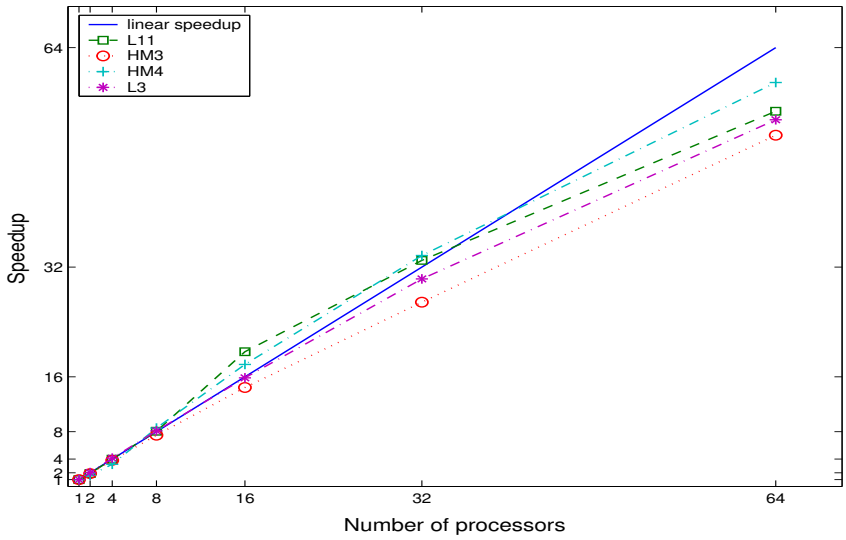

Fig. 3.24: Speedup of the parallel algorithm on some difficult problems

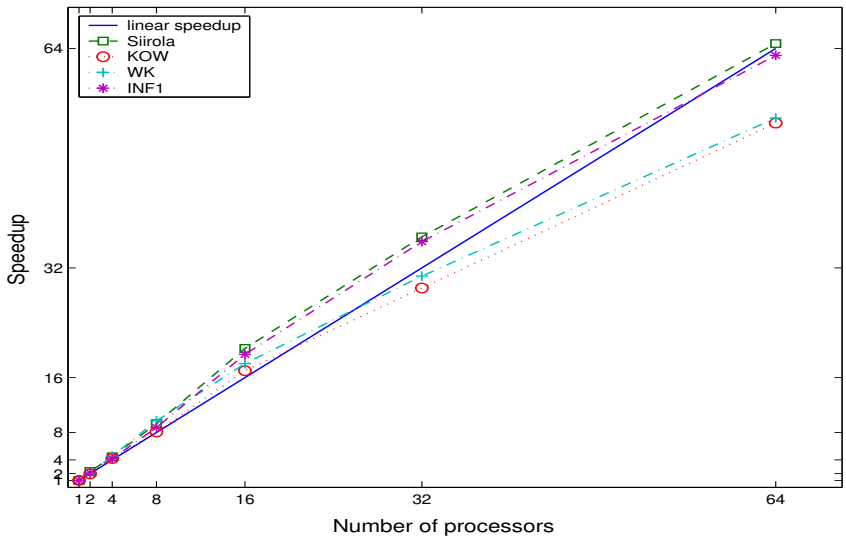

Fig. 3.25: Speedup of the parallel algorithm on some difficult problems

shows the speedup of our parallel algorithm on some test problems. In this case we suppose that the working list is implemented as a simple queue. We see that, for these difficult test problems superlinear speedup is always achieved.

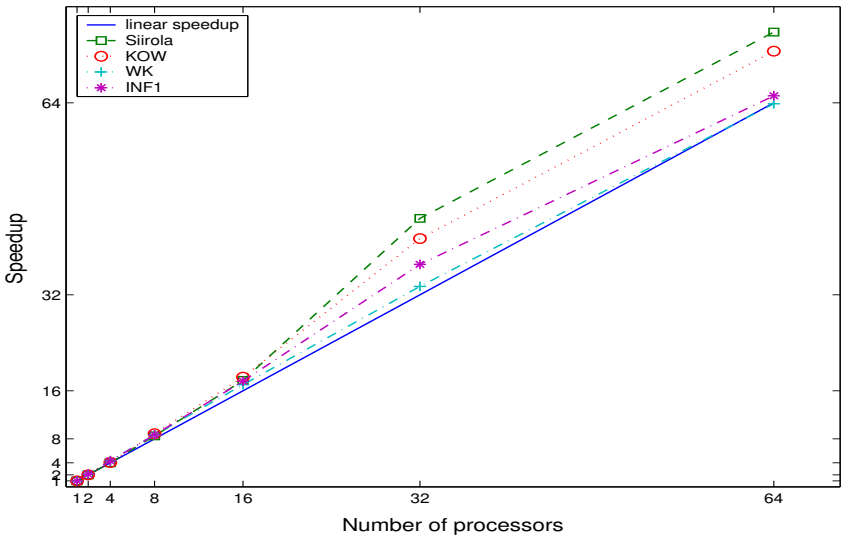

Fig. 3.26: Speedup of the parallel algorithm, the list is implemented as a simple queue

Since the speedup depends on the serial algorithm, a fair comparison of parallel approaches should consider the same serial algorithm for all parallel approaches. This means that, in other to compare approaches, one has to implement all of them. Here we make a comparison with the *Challenge Leadership*, the best of all approaches we have presented.

Comparison with the Challenge Leadership

We have implemented the Challenge Leadership (CL) approach, a fair comparison with the Distributed Management (DM) is possible. Figures 3.27, 3.28, 3.2.4 show the ratio

$$\frac{\text{t\_DM}}{\text{t\_CL}},$$

where t_DM is the time required by the Distributed Management approach on some test problems and t_CL the time required by the Challenge Leadership approach. For many problems we see that there is not a significant difference, still that the Distributed Management achieves an overall better performance. The ratio on medium is contained in [0.8 1.2] and the ratio on hard test problems is contained in [0.9, 1.05]. We see that the ratio on hard problem is even smaller, this may be because for these problems, these two strategies are almost

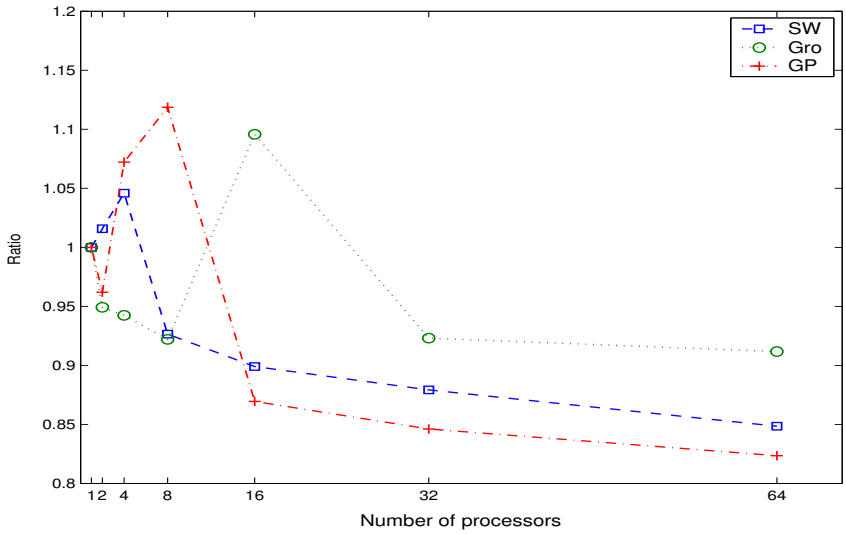

Fig. 3.27: Distributed Management vs. Challenge Leadership on some medium problems

equivalent since there are enough of "good" boxes, consequently processors are always almost busy and there is no need for a frequent load balancing. We can also see that, when the number of processors increases, the Distributed Management is slightly better than the Challenge Leadership. We used up to 64 processors only, may be with more processors the difference between these two strategies could be significant.

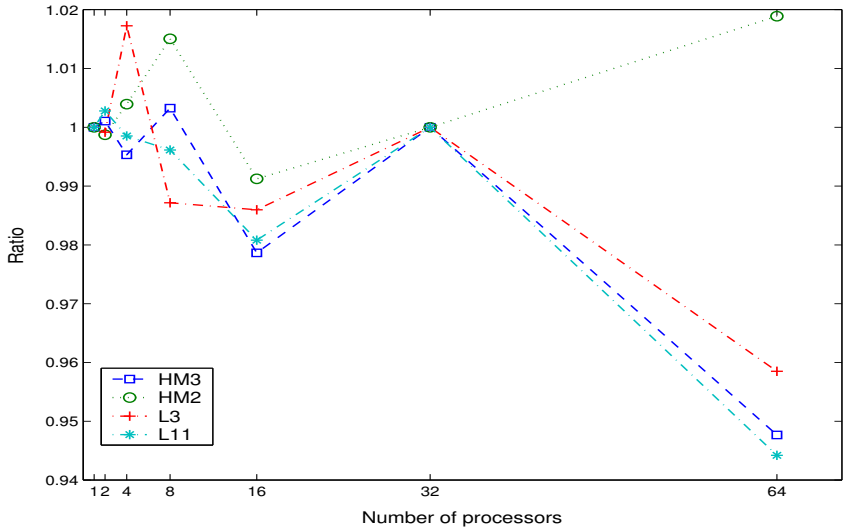

Fig. 3.28: Distributed Management vs. Challenge Leadership on some difficult problems

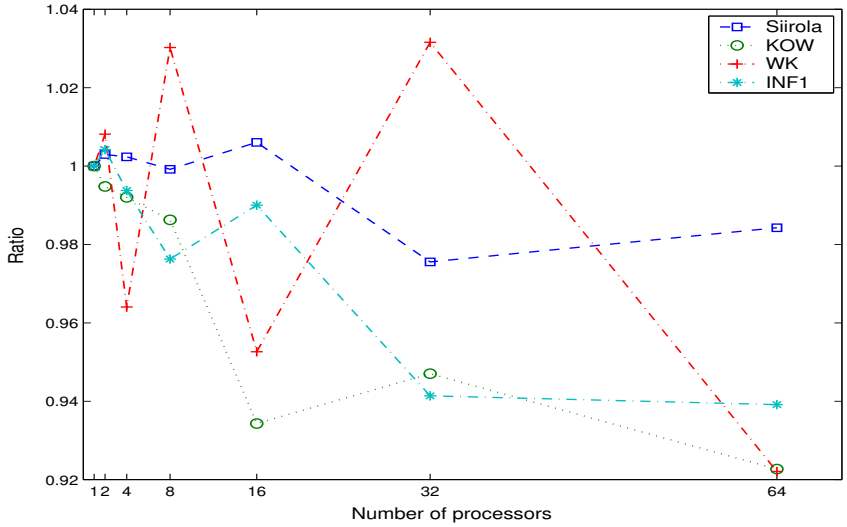

Fig. 3.29: Distributed Management vs. Challenge Leadership on some difficult problems

Appendix A

Considered test problems

A.1 Simple problems

S5($n = 4$, Shekel 5)

$$f(x) = -\sum_{i=1}^{5} = \frac{1}{(x - A_i)(x - A_i)^T + c_i}$$

with [x] = [0, 10]^4, $\epsilon = 10^{-6}$,

$$\mathrm{A} = \begin{pmatrix} 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 \\ 8 & 8 & 8 & 8 \\ 6 & 6 & 6 & 6 \\ 3 & 7 & 3 & 7 \\ 2 & 9 & 2 & 9 \\ 5 & 5 & 3 & 3 \\ 8 & 1 & 6 & 2 \\ 6 & 2 & 7 & 3 \\ 6 & 7 & 3 & 6 \end{pmatrix} \quad \text{and} \quad \mathrm{c} = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.4 \\ 0.4 \\ 0.6 \\ 0.3 \\ 0.7 \\ 0.5 \\ 0.5 \end{pmatrix}$$

The global minimum $f^* \in [-10.153199707210, -10.153199650879]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [4.000036851753, & 4.000037458427] \\ [4.000133096919, & 4.000133461207] \\ [4.000037057807, & 4.000037252354] \\ [4.000133225965, & 4.000133332058] \end{pmatrix}$$

S7($n = 4$, Shekel 7)

$$f(x) = -\sum_{i=1}^{7} = \frac{1}{(x - A_i)(x - A_i)^T + c_i}$$

with [x] = [0, 10]^4, $\epsilon = 10^{-6}$,A and c and in **S5**
The global minimum $f^* \in [-10.402940854942, -10.402940278610]$
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [4.000572392022, & 4.000573448767] \\ [4.000689046287, & 4.000689693693] \\ [3.999489540210, & 3.999489885356] \\ [3.999606062342, & 3.999606263183] \end{pmatrix}$$

S10($n = 4$, Shekel 10)

$$f(x) = -\sum_{i=1}^{10} = \frac{1}{(x - A_i)(x - A_i)^T + c_i}$$

with [x] = [0, 10]$^4, \epsilon = 10^{-6}$,A and c and in **S5**
The global minimum $f^* \in [-10.536410152654, \ -10.536409480641]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [4.000745984918, & 4.000747087330] \\ [4.000592619822, & 4.000593257244] \\ [3.999663227190, & 3.999663575795] \\ [3.999509700323, & 3.999509908542] \end{pmatrix}$$

SHCB($n = 2$, Six-Hump-Camel-Back)

$$f(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$$

with $[x] = [-2, 2]^2, \ \epsilon = 10^{-6}$
The global minimum $f^* \in [-1.031628453614, \ -1.031628453366]$.
Candidates for the global minimizer are

$$\begin{pmatrix} [-0.089842013102, -0.089842013098] \\ [0.712656403010, 0.712656403032] \end{pmatrix}, \begin{pmatrix} [0.089842013098, 0.089842013102] \\ [-0.712656403032, -0.712656403010] \end{pmatrix}$$

BR ($n = 2$, Branin)

$$f(x) = \left(\frac{5}{\pi} - \frac{5.1}{4\pi^2} + x_2 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos x_1 + 10$$

with $[x] = [-5, \ 10] \times [0, \ 15], \ \epsilon = 10^{-6}$
The global minimum $f^* \in [0.397887357729; \ 0.397887361142]$.
Candidates for the global minimizer are

$$\begin{pmatrix} [-3.141718350524, & -3.141466904082] \\ [12.274921258894, & 12.275078374432] \end{pmatrix}, \begin{pmatrix} [3.141574972457, & 3.141610336919] \\ [2.274998780786, & 2.275001215451] \end{pmatrix}$$

$$\begin{pmatrix} [9.424734796677, & 9.424821122815] \\ [2.474998330906, & 2.475001666965] \end{pmatrix}.$$

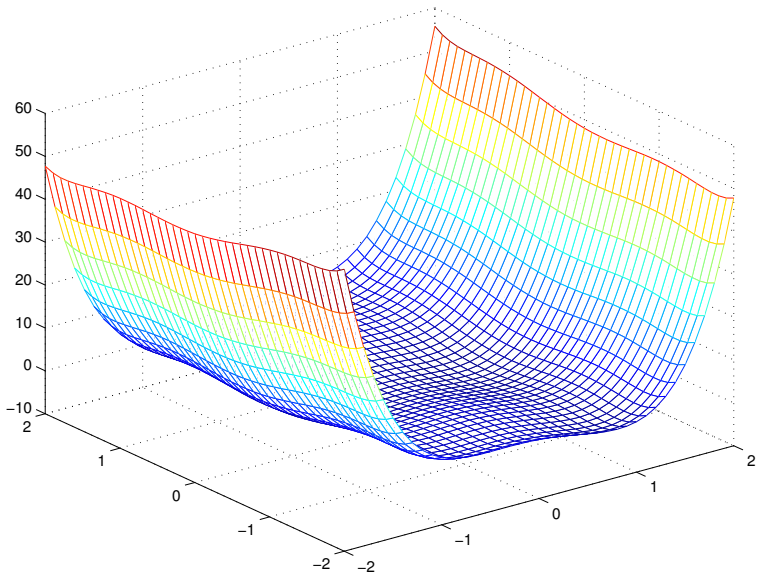

Fig. A.1: The plot of the Six Hump Camel Back function

Ro($n = 2$, Rosenbrock)

$$f(x) = 100(x_2 - x_1^2)^2 + (x_1 - 2)^2$$

with $[x] = [-5,\ 5]^2, \epsilon = 10^{-6}$ The global minimum $f^* \in [0,\ \ 7.551320394136.10^{-8}]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.999988864642,\ \ 1.000011135358] \\ [0.999994813743,\ \ 1.000005186257] \end{pmatrix}$$

L3($n = 3$, Levy 8)

$$f(x) = \sum_{i=1}^{2} (y_i - 1)^2 \left(1 + 10\sin^2(\pi y_{i+1})\right) + \sin^2(\pi y_1) + (y_3 - 1)^2$$

with $y_i = 1 + (x_i - 1)/4, i = 1, \ldots, 3, [x] = [-10,\ \ 10]^3, \epsilon = 10^{-6}$
The global minimum $f^* \in [0.0000000000,\ \ 6.500293555635.10^{-8}]$
The unique verified global minimizer is enclosed in

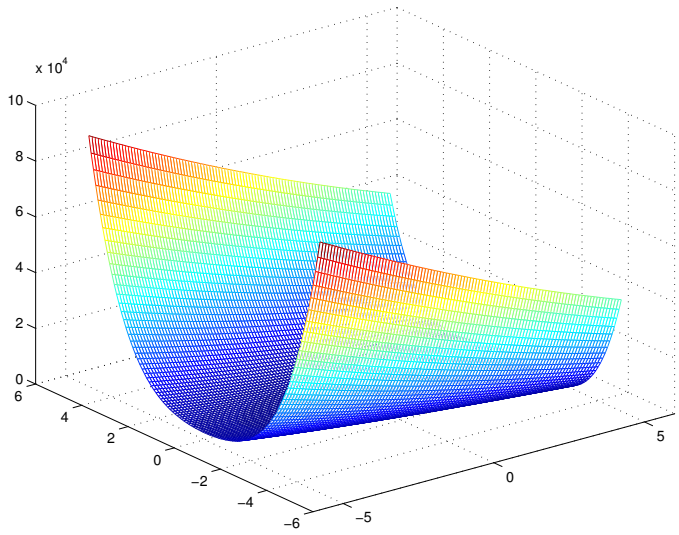

Fig. A.2: The plot of Rosenbrok's function

$$\begin{pmatrix} [0.999999815271, & 1.000000184787] \\ [0.999159280921, & 1.000898023821] \\ [0.999664183571, & 1.000388113611] \\ [0.999790540536, & 1.000288039607] \end{pmatrix}$$

H3($n = 3$, Hartman)

$$f(x) = -\sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{3} A_{ij}\left(x_j - P_{ij}\right)^2\right)$$

with $[x] = [0,\ 1]^3$, $\epsilon = 10^{-6}$

$$A = \begin{pmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1.2 \\ 3 \\ 3.2 \end{pmatrix} \text{ and } \mathrm{P} = \begin{pmatrix} 0.3689 & 0.1170 & 0.2673 \\ 0.4699 & 0.4387 & 0.7470 \\ 0.1091 & 0.8732 & 0.5547 \\ 0.03815 & 0.5743 & 0.8828 \end{pmatrix}$$

The global minimum $f^* \in [-3.862782158846,\ -3.862782136795]$
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.114614313535, & 0.114614363644] \\ [0.555648849192, & 0.555648850752] \\ [0.852546953063, & 0.852546953979] \end{pmatrix}$$

G5 ($n = 5$, Grienwank 5)

$$f(x) = \sum_{i=1}^{5} \frac{x_i^2}{400} - \prod_{i=1}^{5} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

with $[x] = [-500,\ \ 600]^5,\ \ \epsilon = 10^{-6}$
The global minimum $f^* \in [-0.0000000000,\ \ 3.162684336644.10{-}9]$
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [-0.000047126184,\ \ 0.000026346703] \\ [-0.000050237712,\ \ 0.000025987328] \\ [-0.000049887998,\ \ 0.000024425925] \\ [-0.000067131531,\ \ 0.000029231509] \\ [-0.000063538412,\ \ 0.000025042049] \end{pmatrix}$$

A.2 Medium problems

R4($n = 2$, Ratz)

$$f(x) = \sin(x_1^2 + 2x_2^2)\exp(-x_1^2 - x_2^2)$$

with $[x] = [-3, 3]^2,\ \epsilon = 10^{-6}$
The unique global minimum $f^* \in [-0.106891344004,\ -0 : 106891338812]$
Candidates for the global minimizers are

$$\begin{pmatrix} [-3.875919873641E-008,\ \ 3.875919873641E-008] \\ [-1.457522109420,\ \ -1.457522101088] \end{pmatrix}$$

$$\begin{pmatrix} [-3.875919873641E-008,\ \ 3.875919873641E-008] \\ [1.457522101088,\ \ 1.457522109420] \end{pmatrix}$$

L12($n = 10$, Levy 12)

$$f(x) = \sum_{i=1}^{9}(y_i - 1)^2(1 + 10\sin^2(\pi y_{i+1})) + \sin^2(\pi y_1) + (y_{10} - 1)^2$$

with $y_i = 1 + (x_i - 1)/4,\ \ 1, \ldots, 10,\ \ [x] = [-10,\ \ 10]^10,\ \epsilon = 10^{-6}$.
The global minimum $f^* \in [0.000000000000,\ \ 5.022707427890.10^{-12}]$.

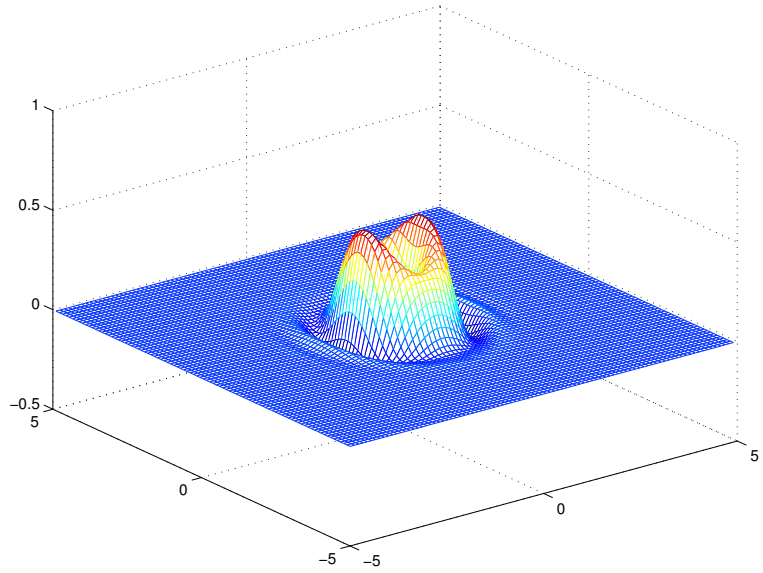

Fig. A.3: The plot of Ratz's function

The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.999999999999, & 1.000000000001] \\ [0.999999999990, & 1.000000000010] \\ [0.999999999989, & 1.000000000011] \\ [0.999999999989, & 1.000000000012] \\ [0.999999996389, & 1.000000003645] \\ [0.999997930503, & 1.000002086249] \\ [0.999992487985, & 1.000007867647] \\ [0.999996881182, & 1.000003658406] \\ [0.999999326761, & 1.000000852685] \\ [0.999999997600, & 1.000000002395] \end{pmatrix}$$

L18($n = 7$, Levy 18)

$$f(x) = \sum_{i=1}^{6}(x_i - 1)^2(1 + \sin^2(3\pi x_{i+1})) + (x_7 - 1)^2(1 + \sin^2(2\pi x_7)) + \sin^2(3\pi x_1)$$

with $[x] = [-5,\ 5]^7$, $\epsilon = 10^{-6}$.
The global minimum $f^* \in [0.000000000000,\ \ 5.415762071898.10{-}12]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.999999999999, & 1.000000000001] \\ [0.999999999988, & 1.000000000012] \\ [0.999999881074, & 1.000000117323] \\ [0.999998120221, & 1.000001807931] \\ [0.999998570159, & 1.000001318913] \\ [0.999999382590, & 1.000000509353] \\ [0.999999889173, & 1.000000097517] \end{pmatrix}$$

G7($n = 7$, Grienwank 7)

$$f(x) = \sum_{i=1}^{7} \frac{x_i^2}{4000} - \prod_{i=1}^{7} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

with $[x] = [-500, \ \ 600]^5, \ \ \epsilon = 10^{-6}$
The global minimum $f^* \in [-0.000000000000; 1.708910790655.10^{-9}]$
The unique global minimizer is enclosed in

$$\begin{pmatrix} [-0.000152632287, & 0.000096304339] \\ [-0.000163024956, & 0.000096174777] \\ [-0.000162823505, & 0.000091817740] \\ [-0.000161711682, & 0.000105549551] \\ [-0.000152903602, & 0.000098344826] \\ [-0.000144957421, & 0.000091601764] \\ [-0.000137798478, & 0.000085371476] \end{pmatrix}$$

G10($n = 10$, Grienwank 10)

$$f(x) = \sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

with $[x] = [-100.5, \ \ 120]^10, \ \epsilon = 10{-}6.$
The global minimum $f^* \in [-0.000000000000; 2.56294.10^{-8}]$.

The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [-0.000042329872, & 0.000041768222] \\ [-0.000035576283, & 0.000035135435] \\ [-0.000040145513, & 0.000040868663] \\ [-0.000029043363, & 0.000029562000] \\ [-0.000021972735, & 0.000022353973] \\ [-0.000017243896, & 0.000017528873] \\ [-0.000013935200, & 0.000014149906] \\ [-0.000011522315, & 0.000011683990] \\ [-0.000009693677, & 0.000009814265] \\ [-0.000008257013, & 0.000008345096] \end{pmatrix}$$

GP($n = 2$, Goldstein Price)

$$\begin{aligned} f(x) = \quad & (1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^3 - 14x_2 + 6x_1x_2 + 3x_2^2)) \times \ldots \\ & (30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_2^2 + 48x_2 - 36x_1x_2 + 27x_2^2)) \end{aligned}$$

with $[x] = [-2; 2]^2, \epsilon = 10^{-6}$.
The global minimum $f^* \in [2.99999953835, \ 3.000000021153]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [-7.092166092395E-011, & 6.674678603178E-011] \\ [-1.000000000048, & -0.999999999962] \end{pmatrix}$$

H6($n = 6$, Hartman 6)

$$f(x) = -\sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)$$

with $[x] = [0, \ 1]^6, \ \ \epsilon = 10^{-6}$

$$\mathrm{A} \ = \ \begin{pmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1.2 \\ 3 \\ 3.2 \end{pmatrix}$$

$$\text{and} \quad \mathrm{P} = \begin{pmatrix} 0.13120.16960.55690.01240.82830.5886 \\ 0.23290.41350.83070.37360.10040.9991 \\ 0.23480.14510.35220.28830.30470.6650 \\ 0.40470.88280.87320.57430.10910.0381 \end{pmatrix}$$

The global minimum $f^* \in [-3.322368011452;\ -3.322368011379]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.201689511002,\ 0.201689511012] \\ [0.150010691821,\ 0.150010691826] \\ [0.476873974209,\ 0.476873974235] \\ [0.275332430494,\ 0.275332430495] \\ [0.311651616600,\ 0.311651616601] \\ [0.657300534065,\ 0.657300534066] \end{pmatrix}$$

S2.14($n = 4$, Schwefel 2.14)

$$f(x) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4$$

with $[x] = [-4, 5]^4$, $\epsilon = 10^{-6}$
The global minimum $f^* \in [0.0000000000000,\ 4.475425668718.10^{-8}]$
Candidates for global minimizers are

$$\begin{pmatrix} [-0.009392075036,\ 0.009317319146] \\ [-0.000907897950,\ 0.000939203138] \\ [-0.002972763767,\ 0.003141232908] \\ [-0.002899169922,\ 0.003141401623] \end{pmatrix}, \quad \begin{pmatrix} [0.002319335937,\ 0.003033963163] \\ [-0.000289916993,\ -0.000221252441] \\ [-0.003233245939,\ -0.003173828125] \\ [-0.003233442004,\ -0.003173828125] \end{pmatrix}$$

$$\begin{pmatrix} [-0.004477072180,\ -0.004272460937] \\ [0.000396728515,\ 0.000447702853] \\ [0.002868652343,\ 0.003184367790] \\ [0.003143310546,\ 0.003184542425] \end{pmatrix}, \quad \begin{pmatrix} [-0.004516072739,\ -0.004272460937] \\ [0.000396728515,\ 0.000451600668] \\ [0.003417968750,\ 0.003612312397] \\ [0.003417968750,\ 0.003612576659] \end{pmatrix}$$

GEO1($n = 3$, The problem from the Geodesy)

$$f(x) = \left(\sqrt{x_2^2 + x_3^2 - 2c_1x_2x_3} - s_1\right)^2 + \left(\sqrt{x_3^2 + x_1^2 - 2c_2x_3x_1} - s_2\right)^2 + \ldots \\ \left(\sqrt{x_1^2 + x_2^2 - 2c_3x_1x_2} - s_3\right)^2$$

with $[x] = [10^{-13},\ 3600] \times [10^{-13},\ 3520]^2$, $\epsilon = 10^{-6}$

$$c = \begin{pmatrix} 0.846735205 \\ 0.928981803 \\ 0.912299033 \end{pmatrix} \quad \text{and s} \quad = \quad \begin{pmatrix} 1871.1 \\ 1592.4 \\ 1471.9 \end{pmatrix}$$

The global minimum $f^* \in [0.000000000000,\ \ 4.064659879814.10^{-8}]$
Candidates for the global minimizers are

$$\begin{pmatrix} [2.292480245974E+003,\ \ 2.292480532764E+003] \\ [3.225046974853E+003,\ \ 3.225047033776E+003] \\ [3.477180122515E+003,\ \ 3.477180155240E+003] \end{pmatrix}$$

$$\begin{pmatrix} [3.575365805357E+003,\ \ 3.575366350973E+003] \\ [3.412155596113E+003,\ \ 3.412155809792E+003] \\ [2.435715337047E+003, 2.435715899727E+003] \\ . \end{pmatrix}$$

GOE2($n = 3$, The problem from the Geodesy)

The same as **GEO1**
with $[x] = [10^{-13},\ \ 8.68] \times [10^{-13},\ \ 9.24] \times [10^{-13},\ \ 8.68]$, $\epsilon = 10^{-6}$.

$$c = \begin{pmatrix} 0.740824038 \\ 0.817119474 \\ 0.737253644 \end{pmatrix} \quad \text{and} \quad \mathrm{s} = \begin{pmatrix} 6.2 \\ 5.0 \\ 6.3 \end{pmatrix}$$

The global minimum $f* \in [0.000000000000,\ \ 1.623435961783.10^{-10}]$.
Candidates for the global minimizers are

$$\begin{pmatrix} [8.369812220149,\ \ 8.369839576033] \\ [8.947975268228,\ \ 8.947982535122] \\ [8.150609828868,\ \ 8.150627639759] \end{pmatrix}, \quad \begin{pmatrix} [4.873871645490,\ \ 4.874651656288] \\ [8.964325384629,\ \ 8.964372850539] \\ [8.118603200621,\ \ 8.118671024373] \end{pmatrix}$$

$$\begin{pmatrix} [8.284519260955,\ \ 8.284618895430] \\ [8.999435873302,\ \ 8.999467348600] \\ [5.288929330144,\ \ 5.289080719743] \end{pmatrix}, \quad \begin{pmatrix} [8.311368014175,\ \ 8.311422879085] \\ [3.271289310727,\ \ 3.271387415143] \\ [8.221033235082,\ \ 8.221046574887] \end{pmatrix}$$

GOE3($n = 3$, The problem from the Geodesy)

The same as **GEO1**
with $[x] = [10^{-13},\ \ 8.0]^3$, $\epsilon = 10^{-6}$

$$c = \begin{pmatrix} 0.766044443 \\ 0.766044443 \\ 0.766044443 \end{pmatrix} \quad \text{and} \quad \mathrm{s} = \begin{pmatrix} 5.0 \\ 5.0 \\ 5.0 \end{pmatrix}$$

The global minimum $f^* \in [0.000000000000,\ \ 1.435824800184 \cdot 10^{-9}]$
Candidates for the global minimizers are

$$\begin{pmatrix} [7.309465389835,\ \ 7.309556607226] \\ [7.309480041821,\ \ 7.309541955399] \\ [7.309493137513,\ \ 7.309528859554] \end{pmatrix}, \quad \begin{pmatrix} [7.309455164529,\ \ 7.309566441765] \\ [3.889171586871,\ \ 3.889445949503] \\ [7.309484162818,\ \ 7.309537442613] \end{pmatrix}$$

$$\begin{pmatrix} [3.889135969558,\ \ 3.889481326344] \\ [7.309473015841,\ \ 7.309548531135] \\ [7.309482573959,\ \ 7.309538972089] \end{pmatrix}, \quad \begin{pmatrix} [7.309463050123,\ \ 7.309558754954] \\ [7.309472945285,\ \ 7.309548860249] \\ [3.889221015008, 3.889397330209] \end{pmatrix}$$

JS($n = 2$, Jennrich-Sampson Problem)

$$f(x) = \sum_{i=1}^{10} (2 + 2i - (e^{ix_1} + e^{ix_2}))^2$$

with $[x] = [-1,\ \ -1]^2,\ \ \epsilon = 10^{-6}$
The global minimum $f^* \in [124.362182355353,\ \ 124.362182355877]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.257825213670,\ \ 0.257825213671] \\ [0.257825213670,\ \ 0.257825213671] \end{pmatrix}$$

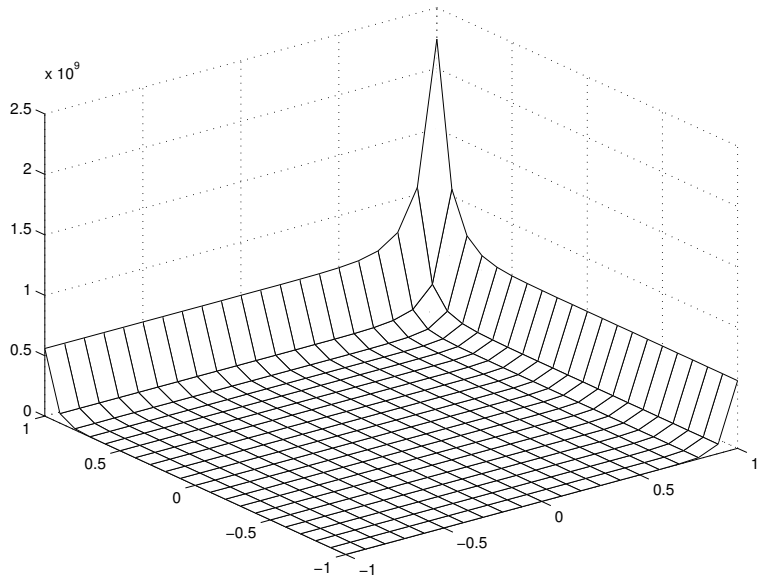

Fig. A.4: The plot of Jennrich-Sampson's function

A.3 Hard problems

S2.7($n = 3$, Schwefel 2.7)

$$f(x) = \sum_{k=1}^{10} \left(\exp\left(\frac{-kx_1}{10}\right) - \exp\left(\frac{-kx_2}{10}\right) - \left(\exp\left(\frac{-k}{10}\right) - \exp\left(-k\right)\right) x_3 \right)^2$$

with $[x] = [0,\ \ 5] \times [8,\ \ 11] \times [0.5\ \ 3]$, $\epsilon = 10^{-6}$
The global minimum $f^* \in [0.00000000000000,\ \ 0.000000132422]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.999760695233,\ \ 1.000238272497] \\ [9.997724926185,\ \ 10.002284236045] \\ [0.999951808024, 1.000048843040] \end{pmatrix}$$

L3($n = 2$, Levy3)

$$f(x) = \sum_{i=1}^{5} i \cos((i+1)x_1 + i) \sum_{j=1}^{5} j \cos((j+1)x_2 + j)$$

with $[x] = [-10,\ \ 10]^2$, $\epsilon = 10^{-6}$
The global minimum $f^* \in [1.867309091505 \cdot 10^2,\ \ 1.867309088310 \cdot 10^2]$.
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [5.4828642057380,\ \ 5.48286420767700] \\ [4.858056878362,\ \ 4.858056879357] \end{pmatrix}, \begin{pmatrix} [-7.708313735502,\ \ -7.708313735496] \\ [-7.083506407653,\ \ -7.083506407650] \end{pmatrix}$$

$$\begin{pmatrix} [-0.800321100472,\ \ -0.800321100471] \\ [-1.425128428320,\ \ -1.425128428319] \end{pmatrix}, \begin{pmatrix} [-7.708313735500,\ \ -7.708313735499] \\ [5.482864206707,\ \ 5.482864206708] \end{pmatrix}$$

$$\begin{pmatrix} [-0.800321100996,\ \ -0.800321099948] \\ [4.858056878605,\ \ 4.858056879115] \end{pmatrix}, \begin{pmatrix} [-1.425128428321,\ \ -1.425128428318] \\ [-7.083506407653,\ \ -7.083506407651] \end{pmatrix}$$

$$\begin{pmatrix} [-7.708313735500,\ \ -7.708313735499] \\ [-0.800321100472,\ \ -0.800321100471] \end{pmatrix}, \begin{pmatrix} [4.858056878859,\ \ 4.858056878860] \\ [-7.083506407652,\ \ -7.083506407651] \end{pmatrix}$$

$$\begin{pmatrix} [-1.425128428320,\ \ -1.425128428319] \\ [5.482864206707,\ \ 5.482864206708] \end{pmatrix}, \begin{pmatrix} [-0.800321100472,\ \ -0.800321100471] \\ [-7.708313735500,\ \ -7.708313735499] \end{pmatrix}$$

$$\begin{pmatrix} [4.858056878859,\ \ 4.858056878860] \\ [5.4828642067070,\ \ 5.48286420670800] \end{pmatrix}, \begin{pmatrix} [4.858056878859,\ \ 4.858056878860] \\ [-0.800321100472,\ \ -0.800321100471] \end{pmatrix}$$

$$\begin{pmatrix} [5.482864206707,\ \ 5.482864206708] \\ [-1.425128428320,\ \ -1.425128428319] \end{pmatrix}, \begin{pmatrix} [5.482864206707,\ \ 5.482864206708] \\ [-7.708313735500,\ \ -7.708313735499] \end{pmatrix}$$

$$\begin{pmatrix} [-1.425128428320,\ \ -1.425128428319] \\ [-0.800321100472,\ \ -0.800321100471] \end{pmatrix}, \begin{pmatrix} [-7.083506407652,\ \ -7.083506407651] \\ [-7.708313735500,\ \ -7.708313735499] \end{pmatrix}$$

$$\begin{pmatrix} [-7.083506407652,\ \ -7.083506407651] \\ [4.858056878859,\ \ 4.858056878860] \end{pmatrix}, \begin{pmatrix} [-7.083506407652,\ \ -7.083506407651] \\ [-1.425128428320,\ \ -1.425128428319] \end{pmatrix}$$

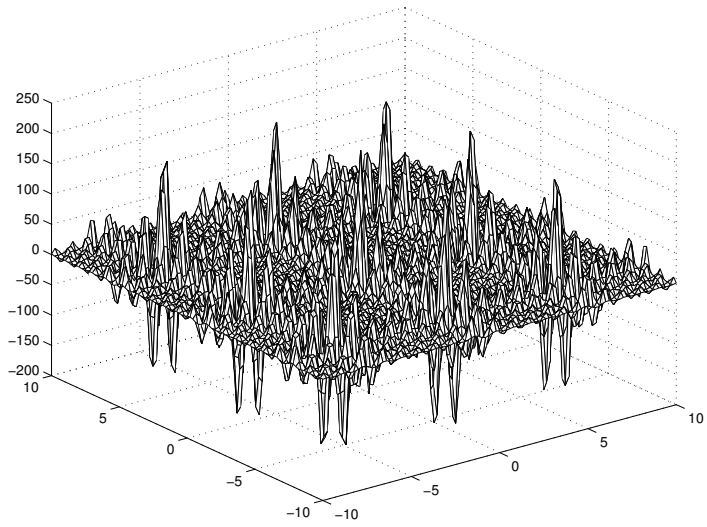

Fig. A.5: The plot of Levy's function

R8($n = 9$, Ratz 8)

$$f(x) = \left(\sin^2\left(\pi \frac{x_1 + 3}{4}\right) + \sum_{i=1}^{8} \left(\frac{x_i - 1}{4}\right)^2 \left(1 + 10\sin^2\left(\pi \frac{x_{i+1} + 3}{4}\right)\right)\right)^2$$

with $[x] = [-10,\ \ 10]^9$, $\epsilon = 10^{-6}$.
The global minimum $f^* \in [0.000000000000,\ \ 0.000000455511]$.
Candidates for the global minimizers are enclosed in

$$\begin{pmatrix} [0.996093750000,\ \ 1.015625000000] \\ [0.937500000000,\ \ 1.015625000000] \\ [0.976562500000,\ \ 1.015625000000] \\ [0.937500000000,\ \ 1.015625000000] \\ [0.937500000000,\ \ 1.015625000000] \\ [0.976562500000,\ \ 1.015625000000] \\ [0.937500000000,\ \ 1.015625000000] \\ [0.976562500000,\ \ 1.015625000000] \\ [-10.000000000000,\ \ 10.000000000000] \end{pmatrix}$$

HM3 (n= 2, Henriksen and Madsen)

$$f(x) = -\sum_{i=1}^{2}\sum_{j=1}^{5} j \sin\left((j+1)x_i + j\right)$$

with $[x] = [-10,\ \ 10]^2$, $\epsilon = 10^{-6}$
The global minimum $f^* \in [-24.062498884345,\ \ -24.062498884330]$
Candidates for the global minimizers are

$$\begin{pmatrix} [-6.774576143440,\ \ -6.774576143438] \\ [-6.774576143440,\ \ -6.774576143438] \end{pmatrix}, \begin{pmatrix} [-6.774576143440,\ \ -6.774576143438] \\ [5.791794470920,\ \ 5.791794470921] \end{pmatrix}$$

$$\begin{pmatrix} [-6.774576143440,\ \ -6.774576143438] \\ [-0.491390836260,\ \ -0.491390836259] \end{pmatrix}, \begin{pmatrix} [5.791794470920,\ \ 5.791794470921] \\ [-0.491390836260,\ \ -0.491390836259] \end{pmatrix}$$

$$\begin{pmatrix} [5.791794470920,\ \ 5.791794470921] \\ [-6.774576143440,\ \ -6.774576143438] \end{pmatrix}, \begin{pmatrix} [-0.491390836260,\ \ -0.491390836259] \\ [-6.774576143440,\ \ -6.774576143438] \end{pmatrix}$$

$$\begin{pmatrix} [5.791794470920,\ \ 5.791794470921] \\ [5.7917944709200,\ \ 5.79179447092100] \end{pmatrix}, \begin{pmatrix} [0.491390836260,\ \ -0.491390836259] \\ [5.791794470920,\ \ 5.791794470921] \end{pmatrix}$$

$$\begin{pmatrix} [-0.491390836260,\ \ -0.491390836259] \\ [-0.491390836260,\ \ -0.491390836259] \end{pmatrix}$$

HM4 ($n = 3$, Henriksen and Madsen)

$$f(x) = \sum_{i=1}^{2}\sum_{j=1}^{5} j \sin\left((j+1)x_i + j\right)$$

with $[x] = [-5,\ \ 5]^3$, $\epsilon = 10^{-6}$
The global minimum $f^* \in [-36.093748326755,\ \ -36.093748326248]$.

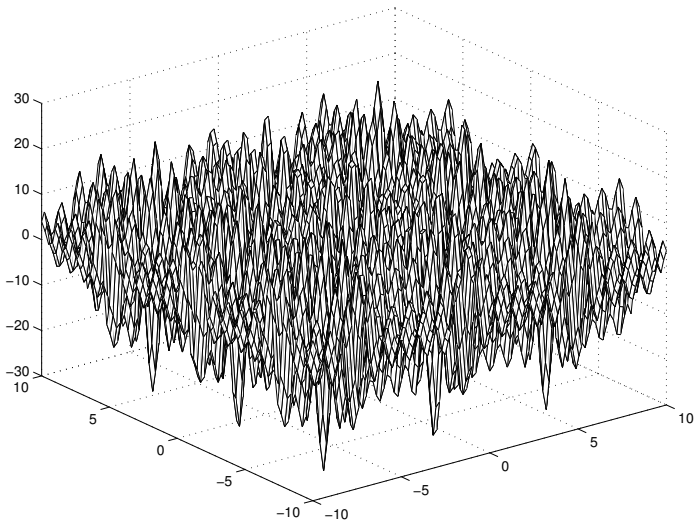

Fig. A.6: The plot of Henriksen and Madsen' function

The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [-0.491390836264, & -0.491390836254] \\ [-0.491390836260, & -0.491390836259] \\ [-0.491390836264, & -0.491390836254] \end{pmatrix}$$

KOW($n = 4$, Kowalik Problem)

$$f(x) = \sum_{i=1}^{11} \left(a_i - x_i \frac{b_i^2 + b_i x_2}{b_i^2 + b_i x_3 + x_4} \right)^2$$

with $[x] = [0,\ \ 0.42]^4]$, $\epsilon = 10^{-6}$

$$c = \begin{pmatrix} 0.19570.1947 \\ 0.1735 \\ 0.1600 \\ 0.0844 \\ 0.0627 \\ 0.0456 \\ 0.0342 \\ 0.0323 \\ 0.0235 \\ 0.0246 \end{pmatrix} \quad \text{and} \quad \mathrm{s} = \begin{pmatrix} 4 \\ 2 \\ 1 \\ 0.5 \\ 0.25 \\ \frac{1}{6} \\ 0.125 \\ 0.1 \\ \frac{1}{12} \\ \frac{1}{14} \\ 0.0625 \end{pmatrix}$$

The global minimum $f^* \in [0.000307140870,\ \ 0.000307616995]$
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [0.192832591586,\ \ 0.192834390270] \\ [0.190819898956,\ \ 0.190850903478] \\ [0.123112693715,\ \ 0.123121520922] \\ [0.135759694776,\ \ 0.135771333126] \end{pmatrix}$$

WK($n = 1$, Kräemer Problem)

$$f(x) = -\frac{p(x)}{q(x)} = -\frac{\sum_{i=0}^{29} p_i x^i}{\sum_{i=0}^{4} q_i x^i}$$

with $[x] = [0,\ \ 64]$, $\epsilon = 10^{-6}$,

$$q = \begin{pmatrix} 0.588286746328683429346629937 6 \cdot 10^{1}1 \\ 0.363467493465600874106423708 7 \cdot 10^{9} \\ 0.996353603100060267502727782 4 \cdot 10^{6} \\ 0.146434177625559953978943514 2 \cdot 10^{4} \\ 1 \end{pmatrix}$$

and

$$p = \begin{pmatrix} 7.629394531250000 \cdot 10^{-6} \\ -1.150369644165040 \cdot 10^{-5} \\ 1.372280530631542 \cdot 10^{-5} \\ -6.579421551577981 \cdot 10^{-6} \\ 1.659054419178573 \cdot 10^{-6} \\ -2.521266665667100 \cdot 10^{-7} \\ 2.505680664436719 \cdot 10^{-8} \\ -1.713721655060476 \cdot 10^{-9} \\ 8.330923088212047 \cdot 10^{-11} \\ -2.935023067946825 \cdot 10^{-12} \\ 7.564193999729689 \cdot 10^{-14} \\ -1.426868803614099 \cdot 10^{-15} \\ 1.954186293985405 \cdot 10^{-17} \\ -1.910400220016202 \cdot 10^{-19} \\ 1.299884226135079 \cdot 10^{-21} \\ -5.995712492310049 \cdot 10^{-24} \\ 1.876066147446556 \cdot 10^{-26} \\ -4.291373306373139 \cdot 10^{-29} \\ 7.622481227988642 \cdot 10^{-32} \\ -1.096397325341554 \cdot 10^{-34} \\ 1.315291857866774 \cdot 10^{-37} \\ -1.344664974747858 \cdot 10^{-40} \\ 1.190815536452828 \cdot 10^{-43} \\ -9.253132527171894 \cdot 10^{-47} \\ 6.374582890249432 \cdot 10^{-50} \\ -3.926998956415952 \cdot 10^{-53} \\ 2.170989219023664 \cdot 10^{-56} \\ -1.142719267106732 \cdot 10^{-59} \\ 3.823185031874960 \cdot 10^{-63} \\ -3.691550884472599 \cdot 10^{-66} \end{pmatrix}$$

The global minimum $f^* \in [-5.129659043375E{-}016,\ \ -4.541716718401E{-}016]$
The unique verified global minimizer is enclosed in

$$\begin{pmatrix} [34.566830008723,\ \ 34.566830519889] \\ [34.566830635070,\ \ 34.566830764922] \end{pmatrix}$$

INF1($n = 2$)

$$f(x) = (x_1 - x_2)^2$$

with $[x] = [-2.0,\ \ 2.5]^2,\ \ \epsilon = 10^{-6}$
The global minimum $f^* \in [0.000000000000,\ \ 1.885928213597E-008]$.
Candidates for global minimizers are

$$[x]^* = [x] \cap \{(x_1,\ \ x_2)\ :\ \ x_1 = x_2\}\,.$$

Siirola($n = 6$)

$$f(x) = 100 \prod_{i=1}^{n} \sum_{j=1}^{5} \left(\frac{j^5}{4425} \cos(j + jx_i) \right) + \frac{1}{n} \sum_{i=1}^{n} (x_i - x_{0,i})^2$$

with $[x] = [x_{0,i} - 20,\ x_{0,i} + 20]^n,\ \ x_{0,i} = 3,\ i = 1, \ldots, n,\ \ \epsilon = 10^{-6}$
The global minimum $f^* \in [-87.241325,\ -87.241324]$
Candidates for the global minimizers are

$$\begin{pmatrix} [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [4.620368,\ 4.620369] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \end{pmatrix}, \begin{pmatrix} [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [4.620368,\ 4.620369] \end{pmatrix}$$

$$\begin{pmatrix} [5.282807, 5.282808] \\ [5.282807, 5.282808] \\ [5.282807, 5.282808] \\ [5.282807, 5.282808] \\ [4.620368, 4.620369] \\ [5.282807, 5.282808] \end{pmatrix}, \begin{pmatrix} [4.620368,\ 4.620369] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \\ [5.282807,\ 5.282808] \end{pmatrix}$$

$$\begin{pmatrix} [5.282806,\ 5.282808] \\ [4.620367,\ 4.620371] \\ [5.282805,\ 5.282810] \\ [5.282804,\ 5.282810] \\ [5.282804,\ 5.282811] \\ [5.282801,\ 5.282813] \end{pmatrix}, \begin{pmatrix} [5.282806,\ 5.282809] \\ [5.282805,\ 5.282810] \\ [4.620366,\ 4.620372] \\ [5.282803,\ 5.282812] \\ [5.282803,\ 5.282812] \\ [5.282799,\ 5.282816] \end{pmatrix}$$

Bibliography

[1] Sonia Berner. *Ein paralleles verfahren zur verifizierten globalen Optimierung*. PhD thesis, Wuppertal University, August 1995.

[2] Sonia Berner. Parallel methods for verified global optimization practice and theory. *Journal of Global Optimization*, 9:1–22, 1996.

[3] Micheal Candev. On the application of an interval algorithm for set inversion. In Götz Alefeld, Andreas Frommer, and Bruno Lang, editors, *Scientific Computing and Validated Numerics*. Akademie Verlag, 1995.

[4] O. Godthaab B. Madsen K. Caprani. Use of a real-value local minimum in parallel interval global optimization. *Interval Computations*, 2:71–82, 1993.

[5] Ole Caprani and Kaj Madsen. An almost embarrassingly parallel interval global optimization method. 2001.

[6] L.G Casados, I. Garcia, and T. Scendes. A heuristic rejection criterion in the interval global optimization algorithms. *BIT*, 2001.

[7] Michel Cosnard and Denis Trystram. *Parallel algorithms and architectures*. International Thomson Computer Press, 1995.

[8] Tibor Csendes. Convergence properties of interval global optimization algorithms with a new class of interval selection criteria. *Journal of Global Optimization*, pages 307–327, 2001.

[9] Tibor Csendes. New subinterval selection criteria for interval global optimization. *Journal of Global Optimization*, 19:307–327, 2001.

[10] Tibor Csendes. Generalized subinterval selection criteria for interval global optimization. *Numerical Algorithms*, 37:93–100, 2004.

[11] R. Hammer M.Hocks U.Kulisch D.Ratz. *C++ toolbox for verified computing.* Springer, 1995.

[12] J. Eriksson. *Parallel Global Optimization Using Interval Analysis.* PhD thesis, University of Umea, Sweden, 1991.

[13] Jerry Eriksson and Per Linderstroem. A parallel interval method implementation for global optimization using dynamic load balancing.

[14] A. Frommer, T. Lippert, B. Medeke, and K.Schilling. *Numerical Challenges in Lattice Quantum Chromodynamics.* Springer, 1999.

[15] William Groop and Ewing Lusk. Reproducible measurements of mpi performance characteristics. Technical report, Argonne National Laboratory, 2000.

[16] Eldon Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, 1992.

[17] P. Hansen, J.L Lagouanelle, and F. Mecine. Comparison between baumann and admissible simplex forms in interval analysis.

[18] Goetz Alefeld Juergen Herzberger. *Introduction to Interval computations.* Academic Press, 1983.

[19] Reiner Horst and Panos M.Pardalos. *Handbook of global optimization.* Kluver Academic Publishers, 1995.

[20] Suiunbek Ibraev. *A new Parallel method for verified global optimization.* PhD thesis, Wuppertal University, October 2001.

[21] Christian Jansson and Jiri Rohn. An algorithm for checking regularity of interval matrices. *SIAM Journal on Matrix Analysis and Applications*, 30:756–776, 1999.

[22] L. Jaulin, J.L Godet, E. Walter, A. Elliasme, and Y. Le Duff. Light scattering data analysis via set inversion. *J.Phys. A:Math*, pages 33–38, 1997.

[23] Luc Jaulin, Michael Kiffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis*. Springer, 2001.

[24] Luc Jaulin and Eric Walter. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29:1053–1064, 1993.

[25] Henriksen T. Madsen K. Use of a depth-first strategy in parallel global optimization. Technical report, Institute for Numerical Analysis, Technical University of Denmark, Lyngby, 1992.

[26] R. Baker Kearfort. *Rigorous Global Search: Continue Problems*. Kluwer Academic Publishers, 1996.

[27] R. B. Kearfott. Interval computations. introduction, uses, and resources. Technical report, University of Southwestern Louisiana, 2001.

[28] Richard W. Kelnhofer. *Applications of interval Methods to Parameter Set Estimation from Bounded-Error Data*. PhD thesis, The Graduate School, Marquette University, 1997.

[29] Y. Lin and M. A. Stadtherr. Advances in interval methods for deterministic global optimization in chemical engineering. In C.A Floudas and P.M Parados, editors, *Frontiers in Global Optimization*. Kluwer Academic, 2003.

[30] Sun microsystem. *C++ Interval Arithmetic Programming Reference*, 2001.

[31] Sun microsystems. *Sun MPI 3.0 Guide*.

[32] M. Milanase and A. Vicino. Estimation theory for nonlinear models and set membership uncertainty. *Automatica*, 27:403–408, 1991.

[33] R.E Moore, E. Hansen, and A. Leclerc. Rigorous methods for global optimization in floudas, c.a., pardalos, p.m.(ed), recent advances in global optimization. *Princeton University Press*, 1992.

[34] Arnold Neumaier. *Interval methods for systems of equations*. Cambrigde University Press, 1990.

[35] Arnold Neumaier. Constraint propagation for univariate quadratic constraints. Technical report, Institut fü Mathematik, Universität Wien, 2005.

[36] University of Illinois. *Introduction to Parallel Computing*. http://pacont.ncsa.uiuc.edu:8900/webct/public/home.pl.

[37] Behrooz Parhami. *Introduction to Parallel Processing*. Plenum Press, 1999.

[38] Dimitri P.Bertsekas and John N.Tsitsiklis. *Parallel and distributed computation*. Prentice Hall, 989.

[39] János D. Pintér. *Global optimization in Action*. Kluwer Academic, 1996.

[40] Coconut projet. *Global constrained optimization and constraint satisfaction*, 2003.

[41] Dietmar Ratz. A nonsmooth global optimization technique using slopes, the one dimensional case. *Journal of Global optimization*, 14:365–393, 199.

[42] Dietmar Ratz. *Automatishe Ergebnisverifikation bei globalen Optimierungsproblemen*. PhD thesis, Karsruhe University, 1992.

[43] Dietmar Ratz. New results on gap-treating techniques in extended interval newton gauss-seidel steps for global optimization. In Bromze, Csendes, Horst, and Parados, editors, *developments in Global Optimization*, pages 55–72. Kluwer Academic Publishers, 1997.

[44] Dietmar Ratz. *Slope Computation and its Application in Nonsmooth Global Optimization*. Shaker-Verlag, 1998.

[45] R.Klatte U.Kulisch A.Wiethoff C.Lawo M. Rauch. *C-XSC, C++ Class Library for extended scientific Computing*. Springer-Verlag, 1993.

[46] H. Ratschek J. Rokne. *New computer methods for global optimization*. Ellis Horwood Limited, 1988.

[47] Oleksandr Romanko. an interior point approach to quadratic and parametric quadratic optimization. Master's thesis, McMaster University, 2004.

[48] Sigfried M. RUMP. Intlab, interval laboratory. *Kluwer Academic*, pages 77–104, 1999.

[49] Sergey P. Shary. A surprising approach in interval global optimization. *Reliable Computing*, 7:497–505, 2001.

[50] D. G. Sotiropoulos and T. N. Graspa. A branch-and-prune method for global optimization: the univariate case. In W. Krämer and J. Wolff von Gudenberg, editors, *Scientific Computing, Validated Numerics, Interval Methods*, pages 215–226. Kluwer Academic/Plenum Publishers, 2001.

[51] Aimo Törn and Antanas Zilinskas. *Global Optimization*. Springer-Verlag, 1988.

[52] Dixon L. C. W. and M. Jha. Parallel algorithms for global optimization. *Journal of Optimization Theory and Applications*, 79, 1993.

[53] Andreas Wiethoff. *Verifizierte globale optimierung auf Parallelrechnern*. PhD thesis, Karsruhe University, 1997.