Parallelization of continuous verified global optimization

N. Revol[*], Y. Denneulin[†], J.-F. Méhaut[‡], B. Planquelle[§]

September 1999

Abstract

An algorithm for verified continuous global optimization, based on interval arithmetic, has been designed by Hansen. This algorithm is of the Branch&Bound type and has an exponential complexity; it thus deserves parallelization. However, this is not straightforward: indeed, it is impossible to predict at compile time which sub-problems will be treated and thus load-balancing can only be performed dynamically.

Our proposal for the parallelization of Hansen's algorithm consists in creating a lightweight process or thread to explore each subinterval and in beginning its execution immediately after its creation. This strategy provides a high level of speculation and enables to virtualize the architecture, since designing the algorithm and porting it to a new architecture are now distinct activities; the load-balancing is also a separate activity. The parallel execution support PM2, developped in Lille and Lyon, handles the concurrent execution of threads and their creation, migration and destruction at a low cost and is our target environment. The implementation of this parallel algorithm is based on the PROFIL/BIAS interval library, preliminary experimental results exhibit superlinear speed-ups.

Keywords: global optimization, interval arithmetic, Hansen's algorithm, parallelization of irregular programs, multithreaded program, dynamic load-balancing.

1 Introduction

It is a challenging problem to be able to determine the global optimum of a function, and to guarantee its optimality, without requiring strong properties for the function. Of course, the smoother the function is and the better the algorithm works. The algorithm we present is much faster if the function to optimize is at least continuously differentiable, it is even faster if the function is twice continuously differentiable and it does not work if the function is not at least piecewise continuous. From a mathematical point of view, characterizations often

[*]N. Revol, ANO, Bât. M3, USTL, 59655 Villeneuve d'Ascq Cedex, France

[†]Y. Denneulin, APACHE, LMC-IMAG, BP 53, 38041 Grenoble Cedex 9, France

[‡]J.-F. Méhaut, projet ReMaP, LIP, École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon, France

[§]B. Planquelle, LIFL, Bât. M3, USTL, 59655 Villeneuve d'Ascq Cedex, France

correspond to a local optimum, unless strong properties (such as convexity) are satisfied by the function to optimize. Classical computational approaches based on probabilistic searches manage to avoid the closest local optimum, but even if the result is improved, it is still not guaranteed to be global. Algorithms based on interval arithmetic obtain verified global optima.

The requirement for guaranteed results is becoming more and more common among automaticians: they need to know whether a system will stay within a safe zone or will exhibit a prohibited behaviour. This requirement has already led them to use interval arithmetic, for instance to solve inversion problems, *i.e.* to determine which values of the parameters guarantee a safe behaviour and which values make the system exit the controllable zone. Indeed, the only ways to obtain guaranteed bounds for a computed result are to compute it either with interval or with exact arithmetic. Interval arithmetic is faster than exact arithmetic and enables to compute elementary functions, however it provides an enclosure of the result and not its exact value.

The basic principle of interval arithmetic [14, 1, 13, 15] consists in replacing every numerical value by an interval containing it; the result of an operation is thus an interval containing every possible numerical result corresponding to every possible numerical operands belonging to the interval operands. The main advantage of this arithmetic is to guarantee that the result Y containing $f(X)$ contains the global optimum of the function f over the interval X: no global information is lost.

There exist some algorithms using interval arithmetic and solving the problem of guaranteed global optimization for continuous functions, cf. [20]. Ratschek and Rokne's discussion of the different algorithms led us to choose Hansen's algorithm [9]: it has a finite execution, provides guaranteed results and uses the more advanced features, which act as accelerating devices. However, interval arithmetic is costlier than the hardware floating-point one. Furthermore, it is a Branch&Bound algorithm, whose worst-case complexity is exponential. An algorithm exhibiting such features deserves parallelization. The main issue in parallelizing such an algorithm is the load-balancing: indeed, it can only be performed dynamically. In this paper we present our proposal for a parallel implementation and some preliminary experimental results.

The parallelization of Hansen's algorithm has been studied by A. Wiethoff [22] and S. Berner [3] for instance. The implementation by A. Wiethoff has a load-balancing strategy based on the number of waiting sub-problems on each processor: the load is measured in terms of the length of the working list local to each processor; the processors are considered to be logically connected via a ring and when a processor is overloaded, it sends a part of this list to a less loaded neighbour. However, this load-balancing strategy does not ensure that the order in which the sub-problems are treated is the best one in terms of the searching heuristic. The parallelization of S. Berner tries to mix centralized and distributed management: centralized management of the sub-boxes has not the previously mentioned drawback. In her implementation, each processor treats its own sub-boxes and when a processor is overloaded (*i.e.* it has too many sub-boxes in its local working list) it sends a part of this working list to the central manager, which redistributes it among idle processors. However, a central manager can constitute a bottleneck on a distributed architecture, especially on a heterogeneous one.

Our proposal is based on a general study of the parallelization of irregular applications. By irregular application is meant an application whose precedence graph depends strongly on the inputs, *i.e.* for a particular instance of the inputs it can be known only dynamically, during the execution [8]. This feature prevents from doing a compile-time parallelization and the load-balancing must be performed on-line. For such applications, we assert that the right programming model consists in a parallelization based on the concurrent execution of dynamically created and destroyed computational activities rather than the message-passing paradigm. An execution support for the parallel execution of irregular applications has been developed in Lille: PM2, Parallel Multithreaded Machine [5, 16], and the load-balancing has been studied for combinatorial optimization applications [6, 7]. In this paper we propose a parallelization of Hansen's algorithm based on threads. Each thread corresponds to the handling of a sub-problem and the threads can be assigned priorities in order to favour the most promising ones according to the search heuristic. Ensuring an execution preserving as much equity as possible on a specific architecture is then a separate task and this distinction enables the programmer to totally virtualize the underlying architecture.

This paper is composed as follows: in the next section the main features of interval arithmetic are briefly introduced and section 3 presents the sequential algorithm for continuous global optimization based on interval arithmetic due to Hansen. In section 4 we develop the paradigm for parallel programming based on threads and PM2, the execution support for multithreaded parallel programs. Then the specific parallelization of Hansen's algorithm is developed and experimental results are given. We conclude and give a sketch of future work in the last section.

2 Interval arithmetic

2.1 Definitions

A (real) interval is a subset of $\mathbb{R}$ of the form

$$X = [\underline{x}; \bar{x}] = \{x \in \mathbb{R} \,/\, \underline{x} \leq x \leq \bar{x}\}$$

where $\underline{x}$ and $\bar{x}$ are real numbers such that $\underline{x} \leq \bar{x}$.
$\mathbb{IR}$ denotes the set of real intervals.

Some useful quantities are the following: if $X = [\underline{x}; \bar{x}]$ is an interval, the lower bound of X is $lb(X) = \underline{x}$, its upper bound is $ub(X) = \bar{x}$ and its center is $mid(X) = \frac{\underline{x}+\bar{x}}{2}$; the width of X is $w(X) = \bar{x} - \underline{x}$ and its magnitude is $|X| = \max\{|x|, x \in X\}$, this last quantity should not be confused with an absolute value.

The arithmetic operations can be extended to interval arguments:

$$\begin{array}{l}
[a;b] + [c;d] = [a+c; b+d] \\
[a;b] - [c;d] = [a-d; b-c] \\
[a;b] * [c;d] = [\min(ac, ad, bc, bd); \max(ac, ad, bc, bd)] \\
1/[a;b] = [1/b; 1/a] \text{ if } 0 \notin [a;b] \\
[a;b]/[c;d] = [a;b] * (1/[c;d]) \text{ if } 0 \notin [c;d]
\end{array}$$

or, to say it in a compact way,

$$X \text{ op } Y = \{x \text{ op } y, x \in X, y \in Y\} \text{ if it is defined.}$$

The algebraic and elementary functions can also be extended to interval arguments, for instance

$$\begin{aligned}
&\sqrt{[4;5]} = [2;\sqrt{5}] \\
&\exp([1;2]) = [e;e^2] \\
&\sin([0;\tfrac{\pi}{4}]) = [0;\tfrac{\sqrt{2}}{2}] \\
&\cos([-2;5]) = [-1;1].
\end{aligned}$$

In order to allow the manipulation of more general expressions, extended intervals have been introduced. An extended interval has one (or two) infinite bound. For instance

$$\begin{aligned}
&[2;3]/[0;1] = [2;+\infty] \\
&[2;3]/[-2;1] = [-\infty;-1] \cup [2:+\infty] \\
&\tan([0;\tfrac{\pi}{2}]) = [0;+\infty]
\end{aligned}$$

The main advantage of interval arithmetic is that every result is *guaranteed*; in other words, if operands are known to belong to intervals X and Y, the result of an operation $\diamond$ between these operands is guaranteed to belong to $X \diamond Y$. To preserve this property with a computer implementation of interval arithmetic, outward rounding is performed; for instance, the computed result of $[a;b]+[c;d]$ is $[\bigtriangledown(a+c);\bigtriangleup(b+d)]$ where $\bigtriangledown$ denotes the rounding towards $-\infty$ and $\bigtriangleup$ denotes the rounding towards $+\infty$.

For the manipulation of functions in several variables, it is mandatory to define interval vectors: X is an interval vector of dimension n, with n components $X_1,\ldots,X_n$ if

$$\begin{aligned}
X = (X_1,\ldots,X_n) &= \{x \in I\!R^n \,/\, x_i \in X_i\} \\
&= \{x \in I\!R^n \,/ lb(X_i) \leq\ x_i \leq ub(X_i)\}.
\end{aligned}$$

$I\!I\!R^n$ denotes the set of interval vectors with n components.
For $n=2$ for instance, an interval vector is a rectangular box. In the following, we will use the word *box* for *interval vector*.
Again, some useful quantities defined on a box are the lower bound $lb(X) = (lb(X_1),\ldots,lb(X_n))$, the upper bound $ub(X) = (ub(X_1),\ldots,ub(X_n))$, the midpoint $mid(X) = (mid(X_1),\ldots,mid(X_n))$ and the width $w(X) = \max\{w(X_1),\ldots,w(X_n)\}$.

Again, arithmetic operations on vectors can be extended in a straightforward way to accept interval vector arguments.

Extending a function defined upon real variables into a function defined upon interval variables is not so straightforward: we will see later that it depends on how the function is expressed. Thus the notion of interval enclosure has been defined: let f be a function of several variables, $f : I\!R^n \to I\!R^m$, F is an *enclosure* of f iff $\forall X \in I\!I\!R^n$, $\forall x \in X, f(x) \in F(X)$. In other words, $F(X)$ contains the range of f over X. It is naturally preferred that $F(X)$ is not too large compared to the range of f over X.

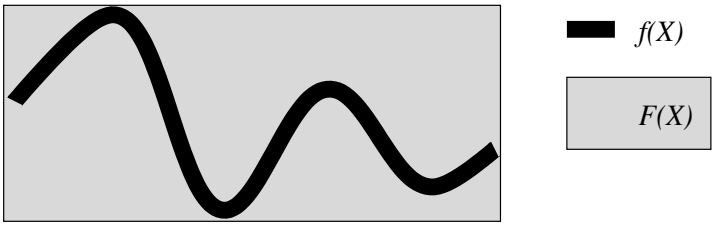

Figure 1: Wrapping effect.

2.2 Properties and problems

Interval arithmetic offers the advantage of providing guaranteed results even when the data are known only with a given accuracy or when the results are computed with a fixed-size representation. However, computing with interval arithmetic presents some drawbacks, besides being slower than the classical arithmetic since every arithmetic operation involves several floating-point operations and since implementations are very often software ones.

The first one is known as *dependency*. This can be observed through the property that multiplication is no more distributive upon addition but only sub-distributive: if A, B and $C \in I\!R$, then $A * (B + C) \subset A * B + A * C$. It can also be observed on the two following examples:

$$[-3; 2] * [-3; 2] = [-6; 9] \neq [-3; 2]^2 = [0; 9]$$

(a square is always positive) and for

$$\begin{array}{rcccl} f & : & I\!R & \rightarrow & I\!R \\ & & x & \mapsto & 4x(x-1) = (2x-1)^2 - 1 \end{array}$$

if we compute the image of $[0; 1]$ using these two formulae, we obtain

$$\begin{array}{l} 4 * [0; 1] * ([0; 1] - 1) = [-4; 0] \\ (2 * [0; 1] - 1)^2 - 1 = [-1; 0]. \end{array}$$

These examples illustrate the decorrelation between the two occurrences of the x variable during the interval evaluation: if $X \in I\!I\!R$, $X * X$ is computed as $\{x * y, x \in X, y \in X\}$ which is indeed very different from $X^2 = \{x^2, x \in X\}$.

The second problem occurring with computation of enclosure function is called *wrapping effect* and is better illustrated by a figure: the image by a function of an interval (vector) is not necessarily an interval vector. However, we have to take an enclosing interval vector as a result since we know how to deal with boxes only.

These problems can be summarized by saying that intervals tend to swell exaggeratedly if computations are performed without particular care. A solution to obtain quite tight bounds is to use different formulae to evaluate a function, unless an expression in which every variable occurs only once is available, since then this expression gives optimal results. Otherwise,

using Taylor-Lagrange expansions of first and second orders can give satisfactory results if the interval arguments are not too wide. These techniques require that the function is $\mathcal{C}^1$ or $\mathcal{C}^2$ and they imply automatic differentiation, or symbolic differentiation at compile time if the compiler is sophisticated enough (numeric differentiation is banned since it is prone to cancellations). To obtain automatically the derivatives of a formula, each intermediate result is replaced by a triple containing this result as first component, the gradient (first order derivative) as second component and finally the Hessian (second order derivative). For instance, in $\mathbb{R}^2$, $x_1^2 + x_2$ is replaced by

$$\begin{pmatrix} x_1^2 \\ \begin{pmatrix} 2x_1 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix} + \begin{pmatrix} x_2 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}$$

and the gradients and Hessians are computed using the classical differentiation rules (chain-rule...).

3 Hansen's algorithm for interval global optimization

E. Hansen has proposed an algorithm using fully the enclosure property of interval arithmetic to solve the problem of continuous global optimization [9]. In what follows the optimization problem is a minimization one. This problem consists in determining f^* the value of the optimum and x^* the point(s) where this optimum is reached:

$$x^* \text{ and } f^* \,/\, : f^* = f(x^*) = \min_{x \in \mathbb{R}^n} f(x) \text{ with } f : \mathbb{R}^n \to \mathbb{R}.$$

Several assumptions are made on the function f and the minimizer x^*:

- a box X_0 is known to contain the global minimizer(s) x^*;
- the global minimizer(s) belongs to the interior of X_0, *i.e.* it does not lie on the boundary;
- the function f is smooth enough: the algorithm presented in the following subsection requires f to be continuous, it works better when $f \in \mathcal{C}^1$ and even better when $f \in \mathcal{C}^2$; it can be adapted without much effort to deal with piecewise continuous functions if the points of discontinuity are known;
- an enclosure F of f is available; if f is $\mathcal{C}^1$, an enclosure G of grad f is available, and if f is $\mathcal{C}^2$ an enclosure H of Hf, the Hessian of f, is available.

3.1 Sketch of the algorithm

Algorithm: Hansen's global optimization
input: $f : \mathbb{R}^n \to \mathbb{R}$ and X_0 box of $\mathbb{R}^n$
output: $[\underline{f}; \bar{f}] \ni f^* = min_{x \in X} f(x)$ and X^* boxes of $\mathbb{R}^n \supset \arg\min_{x \in X} f(x)$
init
  L (list of not yet treated sub-boxes of X) $:= [X_0]$
  compute $\underline{f}$ and $\bar{f}$ lower and upper bounds of f^*
  Res (list of result sub-boxes) $:= \emptyset$
while $L \neq \emptyset$ **loop**
  choose X a sub-box in L and treat X:
    can X be eliminated?
    if X is not eliminated then reduce X
    if X' is small enough then Res $:=$ Res $\bigcup [X']$ else
    cut X' in two sub-boxes X_1 and X_2 and $L := (L \setminus [X]) \bigcup [X_1, X_2]$
end loop

Let us now detail the different procedures constituent of the algorithm.

3.2 The initialization procedure

Thanks to the properties of interval arithmetic, upper and lower bounds on f^* are easily obtained: it suffices to compute $F(X_0)$, an enclosure of the range of f over X_0, to obtain $F(X_0) = [\underline{f}; \bar{f}]$ with $\underline{f} \leq f^* \leq \bar{f}$. A refinement of the upper bound $\bar{f}$ is obtained by computing $F(mid(X_0)) = [a; b]$, then $\bar{f}$ can be replaced by b.

3.3 The reject procedure

Three criteria can be applied to reject a box X, *i.e.* to ensure without further computations that this box X does not contain a global minimizer.

The first one simply consists in comparing the range of f over X with the current upper bound $\bar{f}$ of f^*: if $F(X) > \bar{f}$, then X cannot contain a global optimizer and is rejected. Again, such a range computations is typically what interval arithmetic offers.

The second rejection criterion consists in testing, if f is $\mathcal{C}^1$, whether the gradient of f cancels on X (here the assumption that the optimizer does not lie on the boundary applies), *i.e.* whether grad $f(X) \ni 0$, or rather whether $G(X)$, the enclosure of grad $f(X)$, contains 0. If it does not, then X is rejected. This test is called the monotonicity test.

The last criterion applies if f is $\mathcal{C}^2$. It is called a convexity test and is based on the following property: if X contains a (possibly local) minimizer, then f is locally convex on X. Furthermore, f convex on X is equivalent to the fact that the Hessian $Hf(X)$ is symmetric positive definite, but this is quite complicated to test. Thus a weaker property is used:

$$f \text{ non convex on } X \Leftrightarrow Hf(X) \text{ non SPD } \Leftarrow \exists i \in \{1, \ldots, n\} \,/\, H_{ii}(X) < 0$$

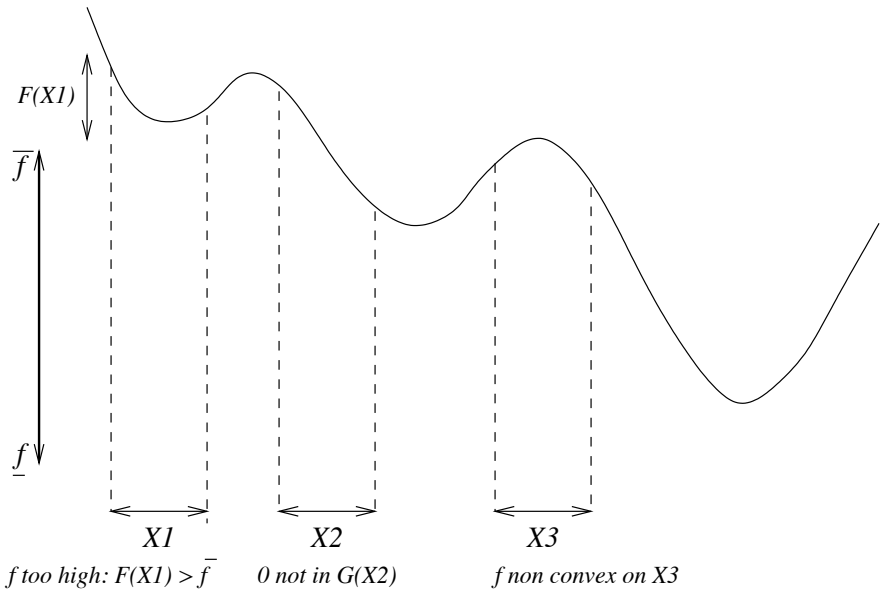

Figure 2: Three rejection criteria.

i.e. this test consists in checking the sign of the diagonal elements of the enclosure $H(X)$ of the Hessian of$f$ over X; if one diagonal element is negative then X is rejected.

3.4 The reduce procedure

If a box X has not been rejected by the previous procedure, then three different techniques are applied to reduce its size. They are based on Taylor expansions of order 1 and 2 and on Newton algorithm and thus can be used if f is $\mathcal{C}^1$ and $\mathcal{C}^2$ respectively.

The first technique uses a Taylor-Lagrange expansion of order 1. If f is $\mathcal{C}^1$, let's denote by $\tilde{x}$ the midpoint of X, $\tilde{x} = mid(X)$, then for any $y \in X$ the following inclusions hold:

$$f(y) \in f(\tilde{x}) + \text{grad } f(X).(y - \tilde{x}) \subset f(\tilde{x}) + G(X).(y - \tilde{x}).$$

This procedure solves the linear inequality

$$Y_1 \subset X? \; / \; f(\tilde{x}) + G(X).(Y_1 - \tilde{x}) \leq \overline{f}$$

and replaces X by Y_1.

The second technique uses a Taylor-Lagrange of order 2: if f is $\mathcal{C}^2$, with the same notations we have

$$\begin{array}{rcl} f(y) & \in & f(\tilde{x}) + \text{grad } f(\tilde{x}).(y - \tilde{x}) + {}^t(y - \tilde{x}).Hf(X).(y - \tilde{x}) \\ & \subset & f(\tilde{x}) + G(\tilde{x}).(y - \tilde{x}) + {}^t(y - \tilde{x}).H(X).(y - \tilde{x}). \end{array}$$

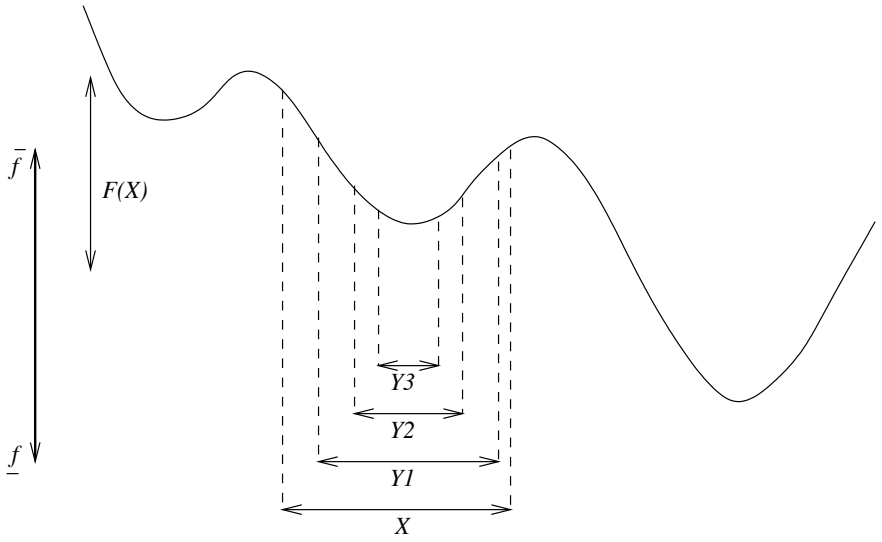

Figure 3: Successive reductions of the box X.

Now the quadratic inequality is solved:

$$Y_2 \subset X? \,/\, f(\tilde{x}) + G(\tilde{x}).(Y_2 - \tilde{x}) +^t (Y_2 - \tilde{x}).H(X).(Y_2 - \tilde{x}) \leq \bar{f}$$

and X is replaced by Y_2.

The last technique derives from a common procedure in floating-point optimization to determine stationary points: if f is $\mathcal{C}^2$, then one step (or some) of Newton algorithm applied to G is performed:

$$Y_3 \subset X? \,/\, G(Y_3) \ni 0.$$

The scalar step of Newton algorithm to solve $x? \,/\, g(x) = 0$ is $x_{k+1} = x_k - g'(x_k)^{-1}.g(x_k)$. In the interval case, each iterate is an interval X_k and the interval step is

$$\begin{array}{l} \text{choose } x_k \in X_k \\ X_{k+1} = Z_{k+1} \cap X_k \end{array}$$

where Z_{k+1} is an enclosure of the solution set Z of the interval linear system $G'(X_k)(x_k - Z) = G(x_k)$. A distinctive feature of the interval Newton step is that it can create gaps in Z_{k+1} (for instance in dimension 1 a gap appears if $G'(X_k)$ contains 0). These gaps can be erased by taking the convex hull of Z_{k+1} or in the contrary be exploited by the split procedure (cf. §3.5). An interesting property of this algorithm is that it enables one to prove the uniqueness of a stationary point: indeed, if X_{k+1} is strictly included in X_k, then it contains a unique stationary point.

3.5 The split procedure

When a box X has been possibly reduced, it is then divided into sub-boxes which will then be processed in the same manner. Different splitting strategies can be found in the literature, such as splitting along the longest side in order to decrease the width of the sub-boxes, or splitting according to the greatest variations of f in the hope that one of the sub-boxes will be quickly rejected, or using the largest gap created by the Newton step.

The number of splittings, and consequently the number of sub-boxes, can also be chosen: according to the implemented version of Hansen's algorithm, the best choice can be to split into 2, 4 or 8 sub-boxes [3, 21].

3.6 Management of the working list

The Moore-Skelboe and the Ichida-Fujii algorithms apply a best first strategy, *i.e.* their working list of boxes X is ordered by non-decreasing values of $lb(F(X))$. Hansen recommends an ordering based either on the width of the boxes (largest first strategy) or on the age of the boxes, when the counting of an age begins at the generation of the box by a splitting (oldest first strategy). These strategies ensure a uniform subdivision of all the boxes not rejected.

The figure 4 illustrates the splittings of the $[-2;2]^2$ box for the Goldstein-Price problem.

3.7 Termination criterion

A box X is accepted as a result box and is thus put in Res, the list of result boxes, rather than in the working list L if one of the two following criteria holds:

- $w(X) < \varepsilon$, *i.e.* X is small enough (and thus rounding errors may prevent further refinements of the bounds of $F(X)$);
- $w(F(X)) < \varepsilon$, *i.e.* the function is very flat on X.

The potential sources of parallelism are the parallel handling of sub-boxes and the parallel computation of the sequence `reject? -- reduce -- split`. For the standard test problems, this `reject? -- reduce -- split` sequence does not deserve parallelization and thus the parallelization effort has been concentrated on the parallel treatment of sub-boxes, as explained in the following section.

4 Parallelization

4.1 Characterization of the problem

Hansen's algorithm belongs to the class of Branch&Bound algorithms; such algorithms are well known in the field of discrete and combinatorial optimization. In particular they are known to have a worst case complexity which is exponential; the complexity of Hansen's

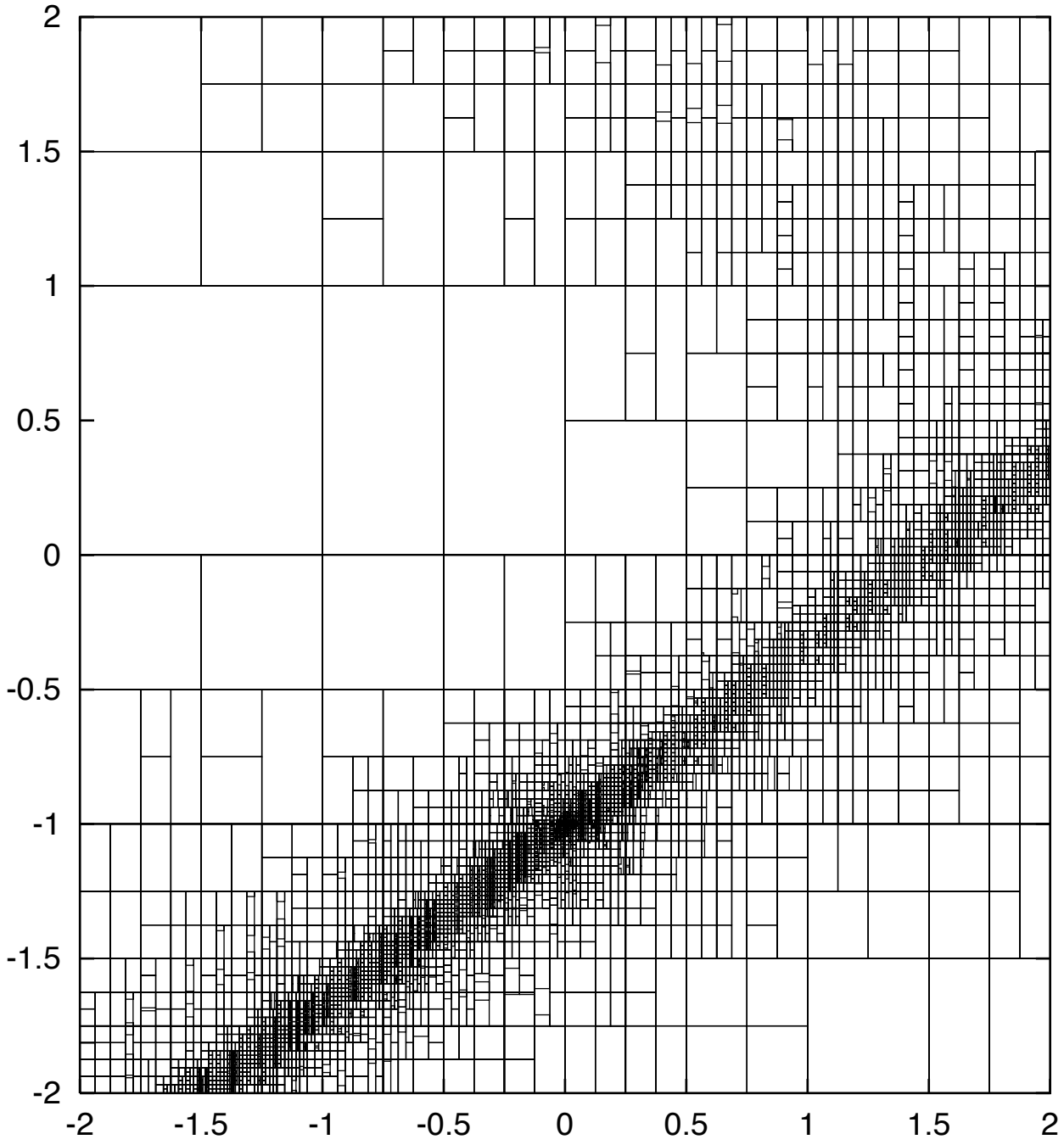

Figure 4: Splittings for the Goldstein-Price problem.

algorithm is indeed exponential in n, the dimension of the problem, and in $1/\varepsilon$ where ε is the required accuracy (cf. termination criterion in §3.7). Furthermore, interval arithmetic operations are slower than their floating-point counterparts since every interval operation involves several floating-point operations. Thus this algorithm deserves parallelization.

This algorithm exhibits the features of an irregular application [8]: its precedence graph is impossible to predict at compile time or at any moment before the execution. Indeed, when a sub-box is handled it is not known in advance whether this box will be quickly rejected or will give rise to long computations.

There are two main models of parallel programming. In the message-passing model, a program consists in a set of heavy computational tasks, with a duration equal to the program's lifetime, and processing their own data; the exchange of data and intermediate results is performed via explicit communications. This model is suitable for regular applications from areas such as dense or structured sparse linear algebra, since the precedence graph is known at compile time and a static parallelization can be done by the compiler or the programmer. Clearly, an irregular application such as Hansen's algorithm does not fulfill these conditions.

The second model is based on a decomposition of the program in tasks, rather than being directed by the data. A parallel program is designed as a set of computational tasks which are handled by lightweight tasks, also called threads. Since these processes are light, their creation, destruction and possibly migration is quite inexpensive; this means that a dynamic parallelization can take place: the load-balancing is performed on the fly, by creating or migrating a task on a under-loaded processor. This model enables the programmer to express an irregular application, in our case Hansen's algorithm, very easily.

4.2 The execution support PM2

PM2 (Parallel Multithreaded Machine)[1] is a runtime environment for the efficient execution of irregular applications developed in Lille and now Lyon by J.-F. Méhaut, R. Namyst and B. Planquelle [5, 16, 19]. The parallel programming paradigm of PM2 is the decomposition of a program into a set of computational tasks, which can be viewed as procedures' calls. These tasks are handled by threads and are launched by LRPCs (Lightweight Remote Procedure Call) when necessary. An example of execution is shown on figure 5: on each node (say, processor) various threads are running. On node X a thread forks a new child thread, which will be executed on the distant node Y, by performing a LRPC; the parent thread may continue its execution and then stops waiting for the result of its child thread (deferred LRPC) or without waiting for the result of its child (asynchronous LRPC).

PM2 provides the efficient handling of threads: creation, context-switching, migration and destruction are realized at a low cost. Two libraries form the core of PM2: MARCEL is the lightweight processes manager and MADELEINE is the communications manager.

PM2 can be seen as an intermediate layer between the machine and the program, insulating the programmer from the specific underlying architecture and thus enabling the virtualization of the architecture. Its other two main goals are to offer a real concurrency of the execution of the various threads composing an application by using preemptive threads

[1] Available at URL `http://www.ens-lyon.fr/ rnamyst/pm2.html`.

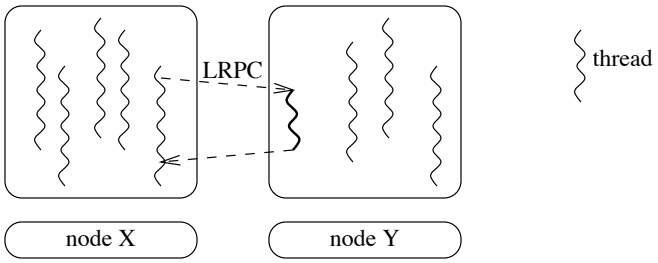

Figure 5: Threads and LRPCs.

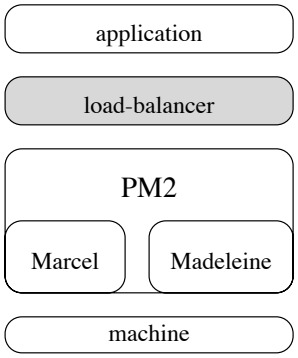

Figure 6: The execution environment PM2.

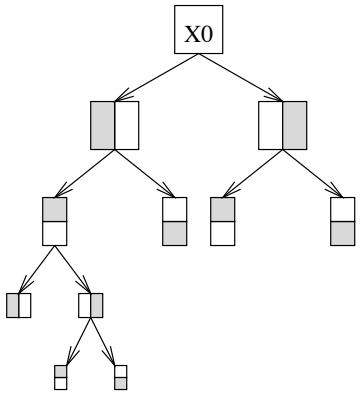

Figure 7: Execution tree of Hansen's algorithm.

and also to guarantee the mobility of the threads even during their execution via a migration mechanism, in order to do dynamic and revisable load-balancing.

Load-balancing is required for irregular applications in order to distribute the computational work between the processors: if the creation of threads were realized by local procedures' calls only, strong differences in terms of computational load could be observed since the precedence graph is unpredictable and thus certainly not regular. Accordingly, this load-balancing must be performed on-line. A load-balancer can be seen as a separate layer between PM2 and the application. Different load-balancers have been studied and implemented by Y. Denneulin during his PhD. [6, 7]. A load-balancer can be global or local, centralized or distributed or hierarchical, it can re-distribute the load according to different strategies such as random work-stealing for instance.

4.3 Parallelization of Hansen's algorithm

Hansen's algorithm contains inherent parallelism since the processing of two disjoint sub-boxes is independent (the only common resource is the value of $\bar{f}$).

If the execution of Hansen's algorithm is represented by a tree where each branch corresponds to the creation of a new sub-box and each node the treatment of this sub-box (cf. figure 7), then it is clear that each node of the tree can be explored independently from its parents and siblings. The parallelization that we propose consists thus in creating a thread to treat each sub-box, a treatment being the sequence `reject? -- reduce -- split`. This approach offers a high level of speculation, *i.e.* it makes it possible to find quickly a good candidate and thus a good value for $\bar{f}$ that enables the rejection of boxes which would have survived a long time in the sequential execution.

Furthermore, it is possible with the implementation of Y. Denneulin to assign priorities to the threads: these priorities are assigned in order to favour the most promising sub-boxes (the "promise" measure being decided by the exploration strategy, *i.e.* by the order chosen

for the working list in the sequential implementation).

The creation of a thread consists solely in the call of a treatment procedure with the coordinates of the box as argument: this argument requires a short storage, indeed of $2n$ floating-point numbers. On the contrary, during the treatment a Hessian is computed and this requires $2n^2$ floating-point numbers: we have thus chosen to switch off the migration mechanism, since migrating a thread can, in our case, take a longer time than completing its execution.

4.4 Experimental results

A preliminary implementation has been realized on a quadri-processor Dell PowerEdge 6300 (Pentium 2 Xeon, 450 MHz). The sequential program we have used is GOP (Global Optimization Program) realized on the PROFIL/BIAS interval library [12] by C. Jansson and O. Knüppel in Harburg-Hamburg, Germany [10, 11] and improved by T. Csendes and M.C. Markót from Szeged, Hungary [4]. We call this a preliminary implementation because we have benefitted from the load-balancing provided by the quadri-processor scheduler and since both the problems and the machine are of quite small sizes. Howver it is the first parallel implementation that is independent of the load-balancer. It validates our approach since the results are excellent. The problems we have experimented are the standard ones, cf. [9, 10, 22]; we present the results for only four typical problems: an easy one which is the six-hump-camel-back problem, denoted by SHCB, two medium-sized problems which are Goldstein-Price (GP) and Hartman 6 (H6) and a more difficult one, Ratz 8 (R8)[2].

[2]The classification is taken from Wiethoff [22].

SHCB: $\min_{x\in X_0} 4x_1^2 - 2.1x_1^4 + 1/3x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$
$X_0 = [-2;2]^2$

GP:
$\min_{x\in X_0} [1 + (x_1 + x_2 + 1)^2 \times (19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)]$

$\times [30 + (2x_1 - 3x_2)^2 \times (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$
$X_0 = [-2;2]^2$

H6: $\min_{x\in X_0} \sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)$
$X_0 = [0;1]^6$

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

$$P = \begin{pmatrix} 0.1312 & 0.1696 & 0.5569 & 0.0124 & 0.8283 & 0.5886 \\ 0.2329 & 0.4135 & 0.8307 & 0.3736 & 0.1004 & 0.9991 \\ 0.2348 & 0.1451 & 0.3522 & 0.2883 & 0.3047 & 0.6650 \\ 0.4047 & 0.8828 & 0.8732 & 0.5743 & 0.1091 & 0.0381 \end{pmatrix}$$

R8: $\min_{x\in X_0} \sin^2\left(\pi \frac{x_1+3}{4}\right) + \sum_{i=1}^{8} \left(\frac{x_i-1}{4}^2\right) \left(1 + 10\sin^2\left[\pi \frac{x_{i+1}+3}{4}\right]\right)$
$X_0 = [-10;10]^9$

The execution times (in ms) in function of the number of threads *simultaneously* handled are shown on figure 8: the sequential vary from 70s to 210s for the medium and difficult problems, the execution time for the easy problem SHCB is very short (about 0.25s). A quantity classically used to measure the performance of a parallel program is the speed-up: it is the ratio between the sequential execution time and the parallel execution time. It depends on p, the number of processors, and is expected to be less or equal to p. The speed-ups in function of the number of threads simultaneously executed are shown on figure 9:

- the speed-up is bad for the easy problem SHCB, which indeed is too fast to need parallelization;
- it is above 3 on the 4 processors for the medium problem GP when enough (6) threads are running; XXX pourquoi ? speculation ?
- it is even better, above 3.5, for the medium problem H6 with at least 4 threads (1 per processor);
- the best speed-up is achieved for the difficult problem R8 with 4 threads (1 per processor); the most remarkable result is the superlinear speed-up (above 4) with a number

of threads greater than 5, even if such speed-up anomalies are known phenomena when parallel Branch&Bound programs are concerned: such superlinear speed-ups are due to the fact that a good value for the upper bound $\bar{f}$ is found quite early in the search and this value enables to reject a lot of sub-boxes which are completely explored in a sequential execution.

Another interesting result is that the speed-up remains at the same level, without decaying, when the number of threads increases (we performed the tests with up to 60 simultaneous threads), which shows that our approach is right.

4.5 Practical problems

We expect to encounter some problems with bigger examples and a purely concurrent execution: if the number of simultaneous threads becomes too important, the scheduler might be overloaded and the memory a limiting resource. It will thus be necessary to create a working list to store the sub-boxes not yet treated. The problems will then be the distributed management of this list: it will have to be balanced in terms of the number of sub-boxes as well as of their quality (the most promising boxes must be fairly distributed). Another solution will be to increase the granularity of a thread (the ratio between its computation time and its communication time) by including in a thread the treatment of a given number > 1 of sub-boxes before creating new threads.

5 Conclusion and future works

Continuous global optimization with guaranteed results is possible thanks to the use of interval arithmetic: Hansen's algorithm solves this problem. This algorithm is of the Branch&Bound type, which has an exponential complexity in the worst-case; furthermore, interval arithmetic operations are slower than their floating-point counterparts. These features make this algorithm a good candidate for parallelization. However, a Branch&Bound algorithm has a unpredictable precedence graph, which prevents from doing any static parallelization. We propose to assign a lightweight process or thread to the handling of each sub-box and to use an execution managing efficiently the concurrent execution of these threads: PM2. A dynamic load-balancer, separate from the application, is used to distribute evenly the computational load among the processors. Our preliminary experimental results illustrate the good and even superlinear speed-ups that can be achieved by this approach.

An on-going work consists in really using PM2 and a load-balancer for our experiments instead of the internal scheduler of the parallel machine, since this device is uncommon on distributed architectures. Furthermore, PM2 has been extended in order to deal with multi-clusters [19]: a multi-cluster is a set of interconnected clusters, each cluster being a set of homogeneous PCs or workstations interconnected by a high-speed network. Clusters can be compared to supercomputers in terms of performances, but they are more widespread and cheaper. Using a multi-cluster should then enable to aggregate even more power at a low cost; however, environments for such architectures are still rare or unsatisfactory. We

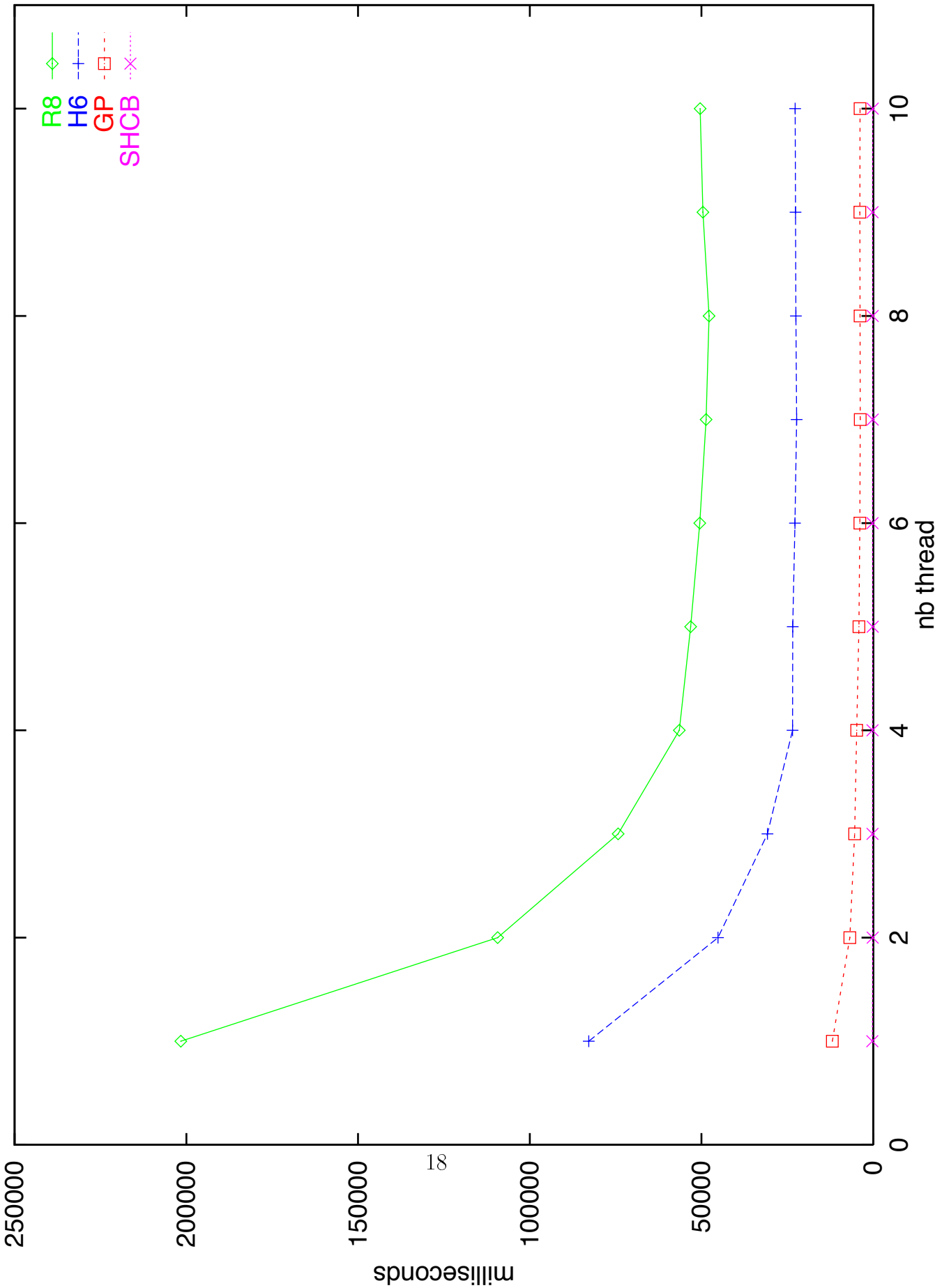

Figure 8: Experimental execution times.

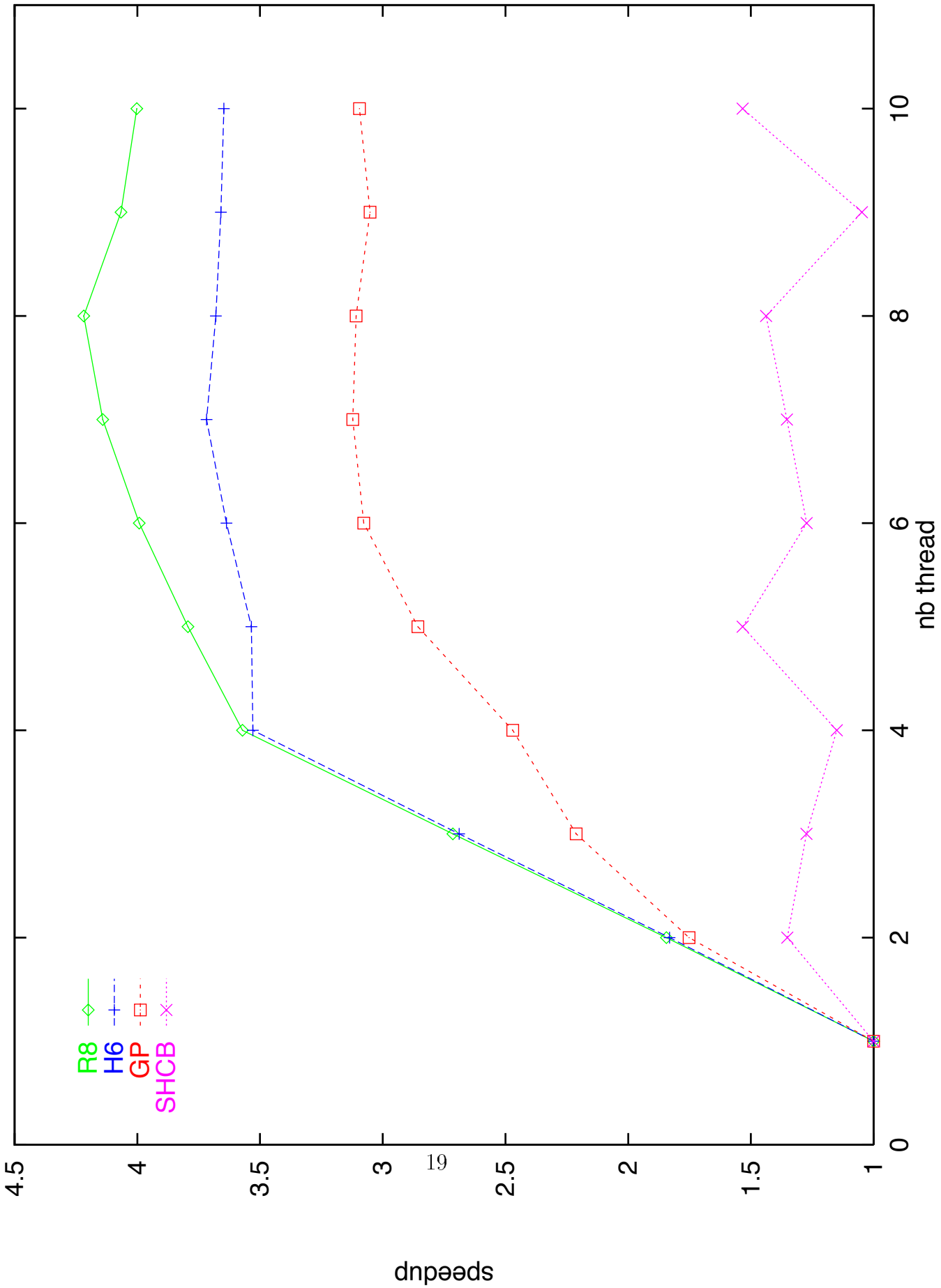

Figure 9: Experimental execution speed-ups.

are now studying the parallelization of Hansen's algorithm on such architectures in order to validate MC-PM2, the extension of PM2 to multi-clusters.

Future work will consist in dealing with constraints: it is not clear, even among the "sequential" community, which strategy is the best. Hansen advocates the use of Fritz-John (or Kuhn-Tucker) formulation in order to remove the constraints and hide them in the objective function; others recommend constraints propagation, *i.e.* testing in the `reject` procedure whether the constraints are satisfied on the current box instead of dealing with a very complex objective function... We also intend to test our implementations with harder problems and with problems from real life: automaticians are at the origin of this work and should provide problems.

Acknowledgements

Thanks to M. Petitot (LIFL) who attracted our attention towards Hansen's algorithm and who worked with us on the understanding of this algorithm.

References

[1] G. Alefeld and J. Herzberger, *Introduction to interval analysis*, 1983, Academic Press.

[2] R. Baker Kearfott, *Rigorous global search: continuous problems*, 1996, Kluwer.

[3] S. Berner, *Ein paralleles Verfahren zur verifizierten globalen Optimierung*, PhD thesis, Wuppertal, Germany, 1996.

[4] A.-E. Csallner, T. Csendes and M.-C. Markót, *Multisection in interval Branch&Bound methods for global optimization*, 1998, submitted.

[5] Y. Denneulin, R. Namyst and J.-F. Méhaut, *Architecture virtualization with mobile threads*, ParCo97, No 12 in Advances in parallel computing, pp 477-484, Elsevier Science Publishers.

[6] Y. Denneulin, *Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin*, PhD thesis, LIFL, Lille, France, 1998.

[7] Y. Denneulin and JF. Méhaut and R. Namyst, *Customizable Thread Scheduling directed by Priorities*, Proc. of Multithreaded Execution Architecture and Compilation (MTEAC 99) held in conjunction of HPCA'5 (High Performance Computer Architecture), pp 35-42, 1999

[8] T. Gautier and J.-L. Roch and G. Villard, *Regular versus irregular problems and algorithms*, Proc. of Irregular 95, LNCS 980, pp 1–25, 1995.

[9] E. Hansen, *Global optimization using interval analysis*, 1992, Marcel Dekker.

[10] C. Jansson and O. Knüppel, *A global minimization method: the multi-dimensional case*, Research report, Harburg-Hamburg, Germany, 1992.

[11] O. Knüppel, *A PROFIL/BIAS implementation of a global optimization algorithm*, Research report 95.4, Harburg-Hamburg, Germany, 1995.

[12] O. Knüppel, *PROFIL/BIAS – a fast interval library*, Computing, vol. 53, number 3-4, pp 277–287, 1994.

[13] U. Kulisch and W.L. Miranker (eds), *A new approach to scientific computation*, 1983, Academic Press.

[14] R.E. Moore, *Interval analysis*, 1966, Prentice Hall.

[15] R.E. Moore (ed), *Reliability in computing – The role of interval methods in scientific computing*, 1988, Academic Press Inc.

[16] R. Namyst, *PM2 : un environnement pour une conception portable et une exécution efficace des applications irrégulières*, PhD thesis, LIFL, Lille, France, 1997.

[17] A. Neumaier, *Interval methods for systems of equations*, 1990, Cambridge University Press.

[18] M.S. Petković and L.D. Petković, *Complex interval arithmetic and its applications*, 1988, Wiley-VCH.

[19] B. Planquelle, J.-F. Méhaut and N. Revol, *Multi-protocol communications and high-performance networks*, EuroPar'99, Toulouse, France. LNCS Springer-Verlag 1685, pp 139–143.

[20] H. Ratschek and J. Rokne, *New computer methods for global optimization*, 1988, Ellis Horwood Ltd.

[21] D. Ratz and T. Csendes, *On the selection of subdivision directions in interval Branch&Bound methods for global optimization*, Journal of Global Optimization, no 7, pp 183–207, 1995.

[22] A. Wiethoff, *Verifizierte globale Optimierung auf Parallelrechnern*, PhD thesis, Karlsruhe, Germany, 1997.