Interval Computations
No 3, 1993

Parallel Interval Global Optimization and Its Implementation in C++

Anthony Leclerc

Using methods which can produce asymptotically accurate lower bounds for the range of values of the function over compact sets, a global solution to the general nonlinear global optimization problem is found. Coding interval arithmetic in C++, the author has designed an ideal bounding mechanism which is capable of producing reliable, "tight", and asymptotically accurate bounds, efficient to compute, applicable to *any* programmable function, easy to generalize and automate. A rigorous algorithm, is presented which produces a list of "boxes" enclosing the set of all global minimizers and an interval trapping the minimum value. For further improvements in efficiency, the algorithm is *parallelized*.

Параллельная интервальная глобальная оптимизация и ее реализация на C++

Э. Леклерк

На основе методов, которые могут дать асимптотически точные нижние границы для множества значений функции на компактном множестве, найдено глобальное решение общей нелинейной глобальной задачи оптимизации. Запрограммировав интервальную арифметику на C++, автор разработал совершенный механизм, который способен получить надежные, <тесные> и асимптотически точные границы. Он эффективен при вычислении, применим к *любой* программируемой функции, легко обобщается и автоматизируется. Представлен строгий алгоритм, с помощью которого можно получить список <боксов>, заключающих в себе множество всех глобальных минимизирующих переменных и интервал, содержащий минимальное значение. Для дальнейшего улучшения эффективности алгоритм *параллелизуется*.

1 Definition of global optimization

Formally, the *global optimization problem* is defined as finding

$$f_* = \min_{x \in X} f(x) \tag{1}$$

where $f : R^n \to R^1$ is a continuous real value *objective function* and $X \subset R^n$ is a compact feasible set. X is often succinctly called *the feasible region*. Since minimizing $f(x)$ is equivalent to maximizing $-f(x)$ this definition sufficiently includes the search for global minima as well as global maxima.

For future discussion, the set of all points for which the objective function possesses a global minimum value shall be called X_*. This is the set which contains all points, x_*, such that $f(x_*) = f_*$. This set is often called the set of *global minimizers*.

2 Computing with intervals

Computing with intervals is computing with sets. Consider the function, $f(x) = x^4 - 8x^2$, with minima at $x = \pm 2$ graphed in Figure 1. If we evaluate f at a point, say, $x = 1$ and over the interval $[3, 4]$ we obtain

$$f(1) = -7 \quad \text{and} \quad f([3, 4]) = [9, 128].$$

From this we know that $f(x) \geq 9$ for all $x \in [3, 4]$ including even transcendental points such as $\pi = 3.14159\ldots$. Since $f(1) = -7$, the minimum value of f cannot occur in the interval $[3, 4]$. This fact has been proven using only two function evaluations.

2.1 Computing with intervals using C++

Today, performing interval arithmetic on a computer with *outward rounding* is easy to achieve. The IEEE standard for binary floating point arithmetic specifies that the ability to round up or down as desired be available in the arithmetic hardware or software. The author has written a fixed precision interval arithmetic package in C and C++ which has been ported to the following systems:

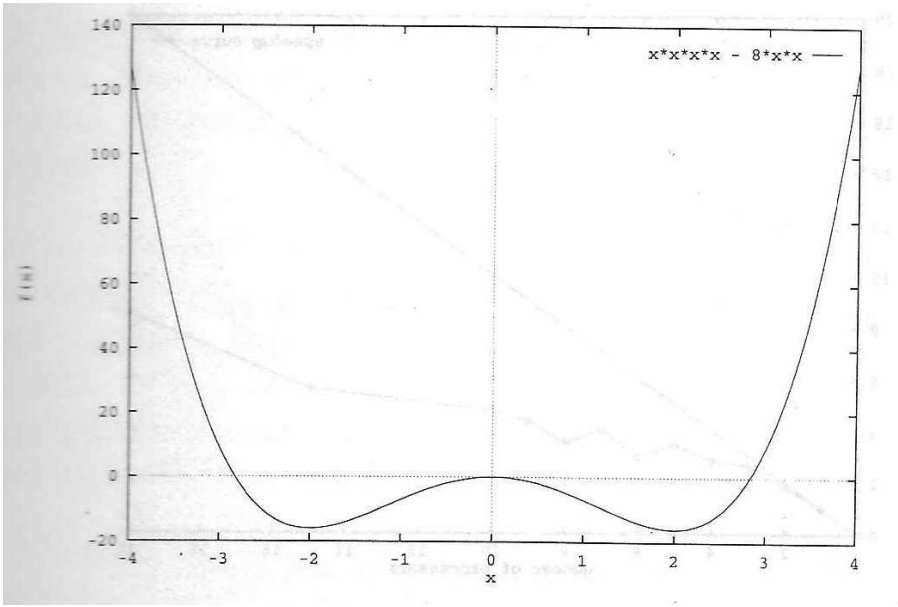

Figure 1: Graph of $f(x) = x^4 - 8x^2$.

- HP300
- SUN3 and SUN4
- IBM-PC
- DECstation

2.2 An example of machine interval arithmetic

Consider the following interval computation of:

$$x(u,t) = \frac{u^2 t}{u^2 + t^2 + 1}$$

where $u = [.1, .3]$ and $t = [.2, .6]$. Evaluating each sub-expression, one obtains:

$$\begin{aligned} u^2 &= [.01, .09] \\ t^2 &= [.04, .36] \\ u^2 t &= [.002, .054] \\ u^2 + t^2 + 1 &= [1.05, 1.45] \\ \frac{u^2 t}{u^2 + t^2 + 1} &= \frac{[.002, .054]}{[1.05, 1.45]} = \left[\frac{.002}{1.45}, \frac{.054}{1.05}\right] \subseteq [.00137, .05143]. \end{aligned}$$

It is hoped that the above example suggests how machine interval arithmetic is performed. For more details, see [7].

3 Reliable global optimization using intervals

Given the ability to compute bounds for the range of f over a set, a simple exhaustive global search algorithm becomes evident. One of the simplest of the bounding methods partitions the initial compact feasible set X into compact subsets, S_i^X. A lower bound on the function value, $F_L(S_i^X)$, over each subset S_i^X is then calculated. In addition, an upper bound on the global minimum thus far, U_{f_*}, is maintained.

Any subset S_i^X where $F_L(S_i^X) > U_{f_*}$ is properly eliminated as not containing a global minimum. This process of partitioning, bounding, and possibly eliminating is continued on successively generated subsets until some stopping criteria is met. The union of the remaining uneliminated sets will contain the set of all global minimizers of f.

All bounding methods, such as branch and bound algorithms [6, 3], covering methods [3], linear lower bound methods [1], Lipschitzian methods [13], bisection methods [3, 14], and interval methods [5, 4, 9, 12, 15, 11], implement the following general algorithm:

1. **Partition** the initial search space into smaller subregions;

2. **Bound** the function (and possibly its derivatives) over the subregions, and

3. **Eliminate** (by using the bounds calculated in Step 2) those subregions which definitely cannot contain a global minimizer.

The union of the remaining *uneliminated* subregions will contain all global minimizers.

The general bounding algorithm uses the *exhaustive* principle. It indirectly searches for a global minimum by exhaustively partioning and "cutting away" all of the feasible space, X, which definitely *cannot* contain a global solution.

Hansen [5, 11] describes an exhaustive interval global optimization algorithm incorporating the following *elimination procedures*:

- **Midpoint test:** Let mX be the feasible midpoint (or any other point) of a sub-box X of the initial search box, B. Also, let $f(mX) = [L_{F_{mX}}, U_{F_{mX}}]$. If f is evaluated over another sub-box Y of B yielding $f(Y) = [L_{F_Y}, U_{F_Y}]$ **and** $L_{F_Y} > U_{F_{mX}}$ then Y cannot contain a feasible global minimizer. Therefore Y can be eliminated.

- **Monotonicity test:** Consider the case in which a box B is certainly strictly feasible. Suppose the gradient g is evaluated over a sub-box X of B. If $0 \notin g_i(X)$ for some $i = 1, \ldots, n$, then the gradient is not zero in X. Therefore the global minimum cannot occur in X, and X can be eliminated.

- **Nonconvexity test:** Again consider a certainly strictly feasible box B and consider a sub-box X of B. If a global solution point, x_* occurs in B, then f must be convex in some neighborhood of x_*. In other words, the Hessian of $f(x)$ must be non-negative definite (positive semi-definite) at x_*. If it can be shown that the Hessian is *not* positive semi-definite anywhere in X, then X can be eliminated.

- **Interval Newton method:** An interval Newton method can be used to eliminate all *or part* of a sub-box X.

4 Distributed algorithm on a network of workstations

A coarse grain parallel global optimization algorithm, based on Hansen's algorithm, is considered. The distributed algorithm has three main steps:

1. **Initialize/startup all processors:**
   - Input initial search box, B, ϵ_x (a box will not be further subdivided if its width is smaller than ϵ_x), and other parameters.
   - Spawn remote processes.
   - Send state information to each "living" process.

2. **Perform Hansen's algorithm in parallel:**
   - Dynamic load balancing (demand driven).
   - Additional load balancing (heuristic for distributing "good" boxes).
   - Broadcasting *new* U_{F_*} (so that all can make sharp midpoint tests).
3. **Terminate all processors:**
   - Detect global termination (centralized algorithm).
   - Compute final solution list (collect all lists).

The steps are defined and discussed in the succeeding sections. Before doing so, the pair of terms *partitioning* and *mapping* are defined in the context of the parallel program. Partitioning refers to the manner in which the input data is divided-up among each of the processors. Mapping is concerned with the particular feasible assignment (with respect to the processor interconnection topology) of processor to process which minimizes communication costs.

A distributed network environment is a fully connected multiprocessor system. Furthermore, the interprocessor communication time is virtually homogeneous. Therefore, in the succeeding algorithm description, the reader can assume that any process can be mapped to any processor, and the mapping issue will not be addressed further. The parallel algorithm is now described.

4.1 Initialize/startup all processors

The initial phase of Hansen's algorithm contains 4 steps:

1. Input initial box, B.
2. Input initial box width tolerance, ϵ_x.
3. Queue the tuple, (B, L_{F_B}), on the *box queue*.
4. Update U_{F_*} .

These first 4 steps are performed only on the main processor, namely P_0. Next, P_0 will attempt to spawn $N-1$ process copies of itself on $N-1$ remote processors, $P_i, 0 < i < N$. P_0 will then wait until it has received from each P_i a *local state message*, *LSM*, indicating the status of the attempted spawn (many errors can occur when attempting to spawn a remote process on a distributed network). Each of the *LSM*s are compiled, along with the sending processor's unique identification number, domain name, and Ethernet address, into a *global state message*, *GSM*. After all $N-1$ *LSM*s have been received, P_0 sends a copy of the *GSM* to all $P_i s$. All processors now have the necessary information to communicate with any other *living* processor involved in the parallel computation.

4.2 Perform Hansen's algorithm in parallel

Once Step 4.1 above has been completed, only P_0 has a box on the box queue. How do the other $N-1$ processors proceed? This brings us to the issue of *dynamic partitioning* and *load balancing*.

4.2.1 Dynamic partitioning and load balancing

Whenever any processor, P_j, has an empty box queue, it begins sending *box request messages*, *BRM*s, to a random $P_i, i \neq j$. If there are boxes available on P_i's box queue, then P_i sends P_j a *box message*, *BM*, containing half of its queued boxes, but no more than *NUMBOXES* (sending arbitrarily large messages is undesirable).

Otherwise, P_i sends back a short *no boxes available message*, *NBM*, indicating that it has no boxes available. If P_j receives a *NBM*, it then sends requests to processors, $P_{(i+1)modN}$, $P_{(i+2)modN}$, $P_{(i+3)modN}$, $\ldots$, $P_{(i+k)modN}$ until it receives a *BM* or until $k = N-1$ (see Section 4.3.1).

This partitioning scheme is dynamic and demand driven. The hope is that by sending half of the workload to each box requesting processor, the work load (number of boxes) among all processors can be balanced.

4.2.2 Broadcasting the new U_{F_*}

As each processor executes Hansen's algorithm in parallel, eventually (assuming there exists a point, $x \in B$ (the initial input box) such that $f(x) < L_{F_B}$) an improved upper bound U_{F_*} on the global minimum will be discovered by a given processor, P_j. At this point, P_j will send this *new* U_{F_*}, NU_{F_*}, to all other processors $P_i, i \neq j$. When a given P_i receives this NU_{F_*} it compares it with its local U_{F_*}. If $NU_{F_*} < U_{F_*}$, P_i updates U_{F_*}. Otherwise, P_i must have received a lower NU_{F_*} from some other processor or calculated a lower U_{F_*} itself during the time it took to receive P_j's NU_{F_*}. In this case, P_i's U_{F_*} is not updated.

4.3 Terminate all processors

With the sequential version of Hansen's algorithm, it was guaranteed that if the first box on the box queue had width less than ϵ_x, then so did all the other remaining queued boxes. However, in the parallel case, if the first queued box on the box queue of given processor, P_i, has width less than ϵ_x, then this does *not* necessarily imply that all the remaining boxes on the $N-1$ other processors' box queues will have width less than ϵ_x.

Indeed, P_i may very well only have found a local minimum. What is P_i to do in this case? If P_i simply prints its output and then terminates, then an uninteresting local solution very likely will be outputted, and moreover, a valuable worker processor will be lost.

The solution, for the moment, is to maintain a second queue, called the *possible solution queue*, *PSQ*, on every processor. Now, if the width of the first queued box on P_i's box queue is less than ϵ_x, then all of the boxes on P_i's box queue are placed on *PSQ*. P_i then behaves as in 4.2.1 for a processor with no queued boxes. Furthermore, whenever P_i determines a *new* U_{F_*}, it checks all boxes on *PSQ* and discards those boxes which fail the midpoint test (see Section 3) using the *new* U_{F_*}. Global termination now becomes a question of detecting when *every* processors' box queue is empty. For the moment, such a state is detected with a simple centralized algorithm. A distributed algorithm using either a ring [2] or a tree [16] would be more efficient and fault tolerant.

4.3.1 Detect global termination

If a given P_i does not receive a *BM* after sending $N-1$ *BRM*s, P_i then sends P_0 a *possible global termination message*, *PGTM*. P_i then waits for either a *BM* or a *terminate message*, *TM*, from P_0. If P_0 receives a *PGTM* and has boxes on its box queue, then P_0 simply sends P_i a *BM*. If P_0 receives a *PGTM* while it has no boxes on its box queue, then P_0 logs P_i's *PGTM*. When P_0 receives $N-1$ *PGTM*s, P_0 sends a *TM* to all other processors. Additionally, if P_0 is sending *BRM*s and receives a *BM*, P_0 must send *BM*s to all processors for which a *PGTM* was logged.

4.3.2 Compute final solution list

When a processor, P_i, receives a *TM*, it prints out all boxes on its *PSQ* and terminates. P_0 does the same as soon as it detects global termination and has sent $N-1$ *TM*s. The final solution list is obtained by combining the output from each terminated processor. As in the sequential version, the union of all the boxes on the final solution list will contain the set of all global minimizers.

4.4 Maintaining reliability with the distributed algorithm

In order to maintain reliability within a distributed environment, the following measures were taken:

- All communication is performed using reliable *socket* datagrams in UNIX. This guarantees that messages sent between processors are not corrupted.
- All possible *signals* are caught. When an error occurs (such as a segmentation fault or bus error) or when the process gets *killed*, the complete contents of the queues are immediately written to a file or sent to another processor. In either case, the algorithm continues as reliably as possible.
- Global termination is determined properly. However, a distributed algorithm using either a *ring* or a *tree* would be more efficient and fault tolerant than the centralized algorithm used.

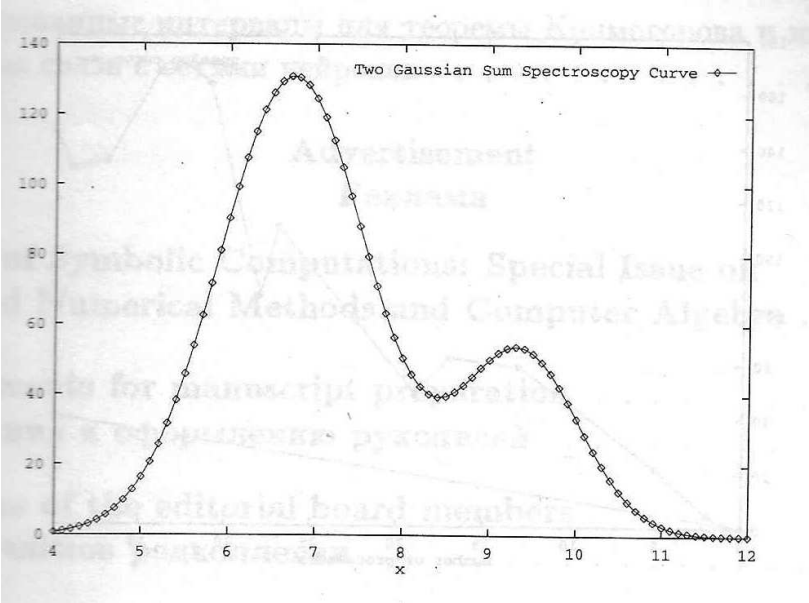

Figure 2: Graph of two Gaussian sum spectroscopy function.

5 Results

The distributed algorithm was tested on the *photoelectron spectroscopy* problem first mentioned in [11].

For this problem, a spectral curve as the sum of two Gaussian functions (see Figure 2) was arbitrarily constructed. The function definition is

$$\begin{aligned} x_i &= 4.0 + 0.1(i+1), \quad i = 1, 2, \ \ldots, \ n \\ y_i &= a_1 e^{-[\frac{x_i - u_1}{s_1}]^2} + a_2 e^{-[\frac{x_i - u_2}{s_2}]^2} \end{aligned}$$

with the constants defined in Table 1.

a_1	=	130.89	a_2	=	52.6
u_1	=	6.73	u_2	=	9.342
s_1	=	1.2	s_2	=	0.97

Table 1: Photoelectron spectroscopy data.

An attempt to "fit" this curve by recovering a_1, a_2, u_1, u_2, s_1, and s_2 was made. Given $n = 81$, (x_i, y_i), and the initial input box, B, defined in Table 2,

B		
a_1	=	$[130, 135]$
a_2	=	$[50, 55]$
u_1	=	$[6, 8]$
u_2	=	$[8, 10]$
s_1	=	$[1, 2]$
s_2	=	$[0.5, 1]$

Table 2: Initial input box for the photoelectron spectroscopy problem.

the task was to minimize f defined as follows:

$$f(a_1, a_2, u_1, u_2, s_1, s_2) = \sum_{i=1}^{n} \left(a_1 e^{-[\frac{x_i - u_1}{s_1}]^2} + a_2 e^{-[\frac{x_i - u_2}{s_2}]^2} - y_i \right)^2.$$

The results were published in [11] with the following timings:

- Single processor time $\approx$ 30 hours;
- 32 processor time $\approx$ 11 minutes.

The parallel algorithm was run on up to 40 processors and achieved superlinear speedup as indicated by Figure 3. Other examples exhibited similar superlinear speedup. In order to explain this superlinear speedup, the progress of the parallel algorithm will be considered in the form of a binary tree.

At the beginning of the algorithm, one is usually given a single initial input box (denoted as the root of the tree). One either eliminates this box or divides it in half yielding two new boxes (depicted as child nodes). Likewise these two new boxes can be eliminated or split. Continuing in this manner, one gradually creates what shall be called a *binary progress tree*.

A portion of one possible binary progress tree is given in Figure 4. The rectangularized regions represent sets of boxes which would be deleted using the current upper bound U_{F_*} on the global minimum. An improved upper bound NU_{F_*} on the global minimum exists within box B_{30}.

In the single processor case, boxes are tested in the order B_1, B_2, ..., B_{31}. The reason for this is the fact that boxes are queued based upon the

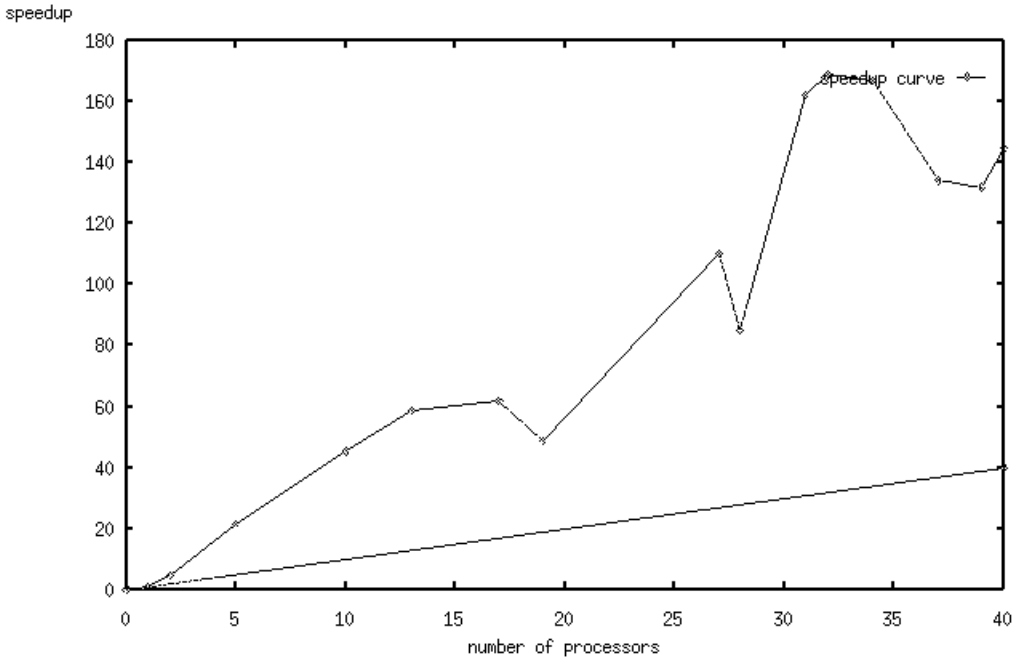

Figure 3: Old speedup graph.

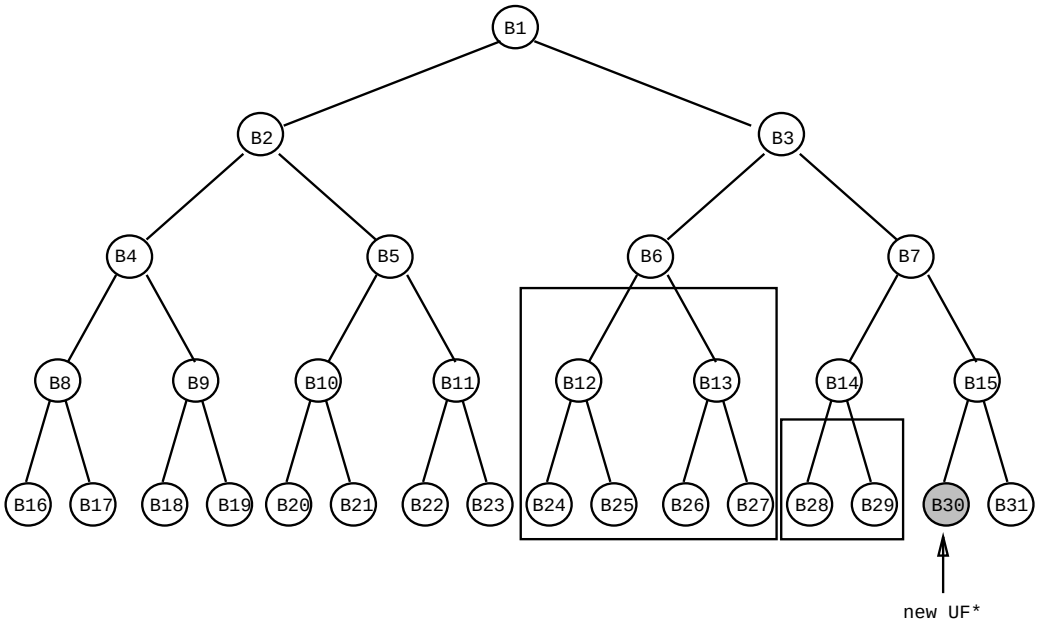

Figure 4: Portion of one possible binary progress tree.

time in which they were generated. This progress amounts to a breadth first search of the entire tree for a "small" enough box containing a solution. Because of this searching strategy, the NU_{F_*} within box B_{30} would require 22 tests before being discovered.

In the two processor case (P_1 initially getting B_2 and P_2 initially getting B_3), each processor would employ a breadth first search on its respective half of the tree. Therefore P_2 would discover the NU_{F_*} in 6 tests (nearly 1/4 the number of tests it took in the single processor case). Furthermore, P_2 would broadcast the NU_{F_*} to P_1 thus allowing P_1 to make sharper midpoint tests earlier and possibly "pruning" other subtrees from consideration. It is this combination of breadth first and depth first searching which is believed to account for the superlinear speedup of the parallel algorithm.

5.1 Two improvements to the distributed algorithm

The superlinear speedup of the distributed algorithm suggests that a better sequential algorithm exists. Indeed, Hansen [5] suggests a superior algorithm based on the observation that it is often the case that the box most likely to contain a global minimum is the one whose lower bound on the function value, L_{F_X}, is lowest.

1. The first improvement to the algorithm was to change the ordering of boxes on the queue from a FIFO manner to a priority queue based on the lowest L_{F_X}. For the spectroscopy problem, this change resulted in a speedup of 78 in the 1 processor case.

2. Secondly, to prioritize the tests *globally*, and not just locally within a processor, each processor now broadcasts its lowest L_{F_X}, which shall be called L_{F_*}. The processor with the largest L_{F_*} requests for "good" boxes from the processor with the lowest L_{F_*}. This change resulted in an additional average speedup of 9 over the older parallel version.

The timing results of the improved algorithm were:

- Single processor time $\approx$ 23 minutes;
- 20 processor time $\approx$ 2.5 minutes.

This improved parallel algorithm was run on up to 20 processors. Approximately 50% of linear speedup was achieved as as indicated by Figure 5.

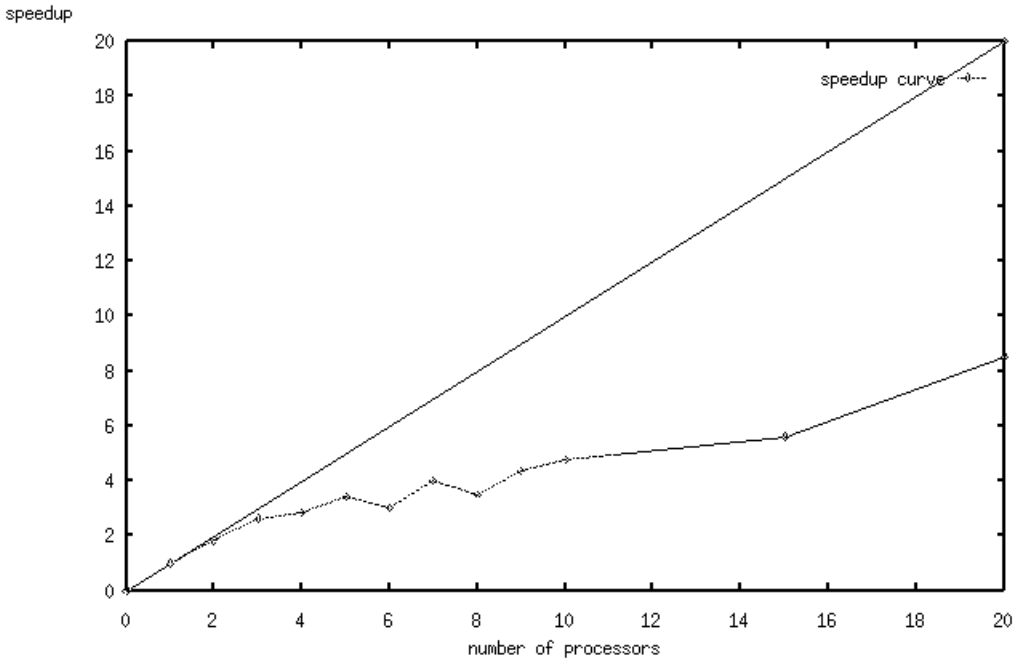

Figure 5: New speedup graph.

References

[1] Bromberg, M. and Chang, Tsu-Shuan. *Linear lower bound approach.* In: "Recent advances in global optimization", Princeton University Press, Princeton, 1992, pp. 200–220.

[2] Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M. *Derivation of a termination detection algorithm for distributed computations.* Information Processing Letters **16** (5) (June 1983), pp. 217–219.

[3] Evtushenko, Yu. G., Potapov, M. A., and Korotkich, V. V. *Covering methods.* In: "Recent advances in global optimization", Princeton University Press, Princeton, 1992, pp. 274–297.

[4] Hansen, E. R. *Global optimization using interval analysis — the multi-dimensional case.* Numer. Math. **34** (1980), pp. 247–270.

[5] Hansen, E. R. *Global optimization using interval analysis.* Marcel Dekker, Inc., New York, 1992 (to be published).

[6] Rinnooykan, A. H. G. and Timmer, G. T. *Argument for the unsolvability of global optimization problems.* In: "New methods in optimization and their industrial uses", Birkhäuser Verlag, Basel, 1989, pp. 133–155.

[7] Moore, R. E. *Methods and applications of interval analysis.* In: "SIAM Studies in Applied Mathematics", SIAM, Philadelphia, 1979.

[8] Moore, R. E. (ed.) *Reliability in computing.* Academic Press, 1988. (See especially the papers by Hansen, E. R., pp. 289–308, Walster, G. W., pp. 309–324, Ratschek, H., pp. 325–340, and Lodwick, W. A., pp. 341–354.)

[9] Moore, R. E. *Global optimization to prescribed accuracy.* Computers Math. Applic. **21** (1991), pp. 25–39.

[10] Moore, R. E. *Interval tools for computer aided proofs in analysis.* The IMA Volumes in Mathematics and Its Applications **28** (1991), pp. 211–216.

[11] Moore, R. E., Hansen, E. R., and Leclerc, A. P. *Interval Global Optimization.* In: "Recent Advances in Global Optimization", Princeton University Press, Princeton, 1992, pp. 321–342.

[12] Moore, R. E. and Ratschek, H. *Inclusion functions and global optimization II.* Mathematical Programming **41** (1988), pp. 341–356.

[13] Pintér, J. *Lipschitzian global optimization.* In: "Recent Advances in Global Optimization", Princeton University Press, Princeton, 1992, pp. 399–432.

[14] Ratschek, H. *Inclusion functions and global optimization.* Mathematical Programming **33** (3) (1985), pp. 300–317.

[15] Ratschek, H. and Rokne, J. *New computer methods for global optimization.* Ellis Horwood and John Wiley, 1988.

[16] Topor, R. W. *Termination detection for distributed computations.* Information Processing Letters **18** (1) (January 1984), pp. 33–36.

The College of Charleston
Department of Computer Science
5771 Avenue Chateau du Nord
Columbus, OH 43229
USA