Interval Computations
No 3(5), 1992

PARALLEL ALGORITHMS FOR GLOBAL OPTIMIZATION

Tom Henriksen and Kaj Madsen

Two versions of the classical interval method for global optimization are paralleled. Experiments with examples in up to 10 variables and using up to 32 parallel processors indicate that the methods parallel quite efficiently.

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ ГЛОБАЛЬНОЙ ОПТИМИЗАЦИИ

Т. Хенриксен, К. Мадсен

Предложены распараллеленные варианты двух версий классических интервальных методов глобальной оптимизации. Результаты экспериментов на примерах, содержащих до 10 переменных и использующих до 32 параллельных процессоров, показывают достаточную эффективность распараллеливания.

1. Introduction

Many technical and economic problems can be formulated as mathematical programming problems, i.e. as the minimization of a continuous non-linear function $f : D \to R$ where $D \subseteq R^n$. Very often the function f has several local minima only one of those being interesting however, namely the smallest one:

$$f^* = \inf\{f(x) | x \in D\} \tag{1}$$

f^* is the global minimum, and the problem of finding f^* and the points where it is attained is called *global optimization.*

We use the classical interval method for global optimization see e.g. [4-6]. It is a branch and bound type method. At any stage of the procedure we have a finite number of

subsets S_i of D

lower bounds, $LB(S_i)$, on $\min\{f(x)|x \in S_i\}$

upper bounds, $UB(S_i)$, on $\max\{f(x)|x \in S_i\}$

with the property any global minimizer is contained in the union of all subsets D_i, i.e.

$$D \supseteq \cup_{i=1}^{p} S_i \supseteq \{x \in D | f(x) = f^*\}. \tag{2}$$

A subset S_i is an interval and is also called a *box.* The set of boxes $\{S_i\}$ is called *the candidate* set and denoted by S. Now the aim is to reduce the candidate set in an attempt to obtain equality in the second inclusion of (2).

We present two parallel versions of the classical interval method. The difference between the strategies lies in the order in which members from the candidate set are considered. In the first method we always consider that candidate which has the smallest lower bound. This will require some sorting of the candidate set, for efficiency reasons. It is denoted the *breadth-first* method.

In the second method the candidates are treated in a *depth-first* order: If treatment of a candidate leads to two subsets then one of there is treated immediately (unless it can be deleted or the stopping criterion is met), and the other is added to the candidate set. In this method we assume an estimate of the minimum, to be given. This could be found by a traditional optimization method for finding local minima. It should be noted that the time used to do a traditional optimization normally is negligible compared to the time used by the interval method. The depth-first strategy may be considered as a method for proving globalness of the local solution found, or to disprove it and then finding the global minimum.

Both strategies are parallised using a central method on a Multiple Instruction Multiple Data (MIMD) machine. A *central* processor has

the overall control and the other processors (*the slaves*) are doing the treatment of candidates.

2. The sequential interval methods

We first sketch the classical interval method for global optimization.

Let *fbound* be an upper bound on f^*, i.e. we always have the inequalities

$$m_s \equiv \min\{LB(S_i) | S_i \in S\} \leq f^* \leq \text{ fbound.} \tag{3}$$

Furthermore, we keep a set of final results R, i.e. intervals which fulfill the stopping criterion. Such intervals are not inserted into S.

Then the model algorithm, is the following

initialize→ S, fbound
while (S is non-empty) **do**
    remove-smallest $(S) \rightarrow S_i$
    reduce-or-subdivide (S_i) →result, fbound (4)
    insert result into S or R
    for $S_i \in S$ **do if** $LB(S_i)$ >fbound **then** $S = S \setminus S_i$
end

Of course the **for**-statement needs not always search the whole of S.

In (3) any known function value can be used as *fbound* since $f^* \leq f(x)$ for any $x \in D$. Thus *fbound* can be reduced every time a better function value is found.

Now let F be an interval extension of f. Then the details in (4) are the following:

initialize: $S = \{D\}$, fbound=max$\{F(D)\}$

reduce-or-subdivide($\mathbf{X}$):

if *monotone* **then** result=mon($\mathbf{X}$)
else if *stationary* **then** result=Newton($\mathbf{X}$) (5)
else
subdivide ($\mathbf{X}$) $\rightarrow$ $\mathbf{X1}, \mathbf{X2}$
result = $\{\mathbf{X1}, \mathbf{X2}\}$

After a successful *monotonicity* test an interval extension $\mathbf{F}' = (F_1 ..., F_n)^T$ of the gradient f' is used to reduce $\mathbf{X}$. $mon(\mathbf{X})$ is the result of this reduction.

The *stationary* test tells if a version of the interval Newton method [5] can be used to search for a zero of the gradient f' (a stationary point). The result is $Newton(\mathbf{X})$.

Subdivide: Here $\mathbf{X}$ is divided into two, for instance by halving the side of largest width.

The difference between the two strategies presented in this paper lies in the implementation of the remaining two details.

The breadth-first algorithm. This is the traditional interval method.

remove-smallest: The box S_i with smallest value of min$\{F(S_i)\}$ is removed from S.

The value of *fbound* may be updated by

fbound := min{ fbound,min$\{UB(\mathbf{Z}) \mid \mathbf{Z} \in$ result$\}\}$
or fbound := min{ fbound, $f(z)\}$ (6)

where $z \in \mathbf{Z}$ can be chosen appropriately using the (known) gradient information.

The depth-first algorithm. Here the idea is to finish the work on one box before starting to subdivide the others. The advantage of this strategy is that the number of elements in S is kept much smaller, and,

as we shall see, it may be better suited for paralleling. The disadvantage is that the value of *fbound* may not be decreased so quickly because the boxes that give the small values may not be examined until late in the calculation. Therefore the total amount of computation needed here may be larger. In order to compensate for this disadvantage we combine the interval method with a traditional minimization method: We find a local minimum using the traditional method and use the function value found as the initial *fbound*.

To summarize, the remaining details of the depth-first algorithm are:
remove-smallest: The box S_i which has last entered S is removed.
fbound is initialized as described above and possibly updated using (6).

If we fix the stopping criterion then we can compare the amount of work in the two methods. Let the criterion be

$$\begin{array}{ll} \textbf{if } (w(S_i) < \epsilon \text{ and } LB(S_i) \leq m_s) & \textbf{then } (\text{insert } S_i \text{ in } R) \\ & \textbf{else } (\text{insert } S_i \text{ in } S) \end{array} \tag{7}$$

Theorem. *Assume (8). If the local mininimizer used in the depth-first algorithm finds the global minimum f^* then the treated are the same in the two methods.*

Proof. Both methods give the same treatment to a specific box S_i. It is only the order of treating the candidates which differs. However, since we may have *fbound* $> f^*$ during the execution of the breadth-first method, a box $\mathbf{X}$ with $LB(\mathbf{X}) > f^*$ may be inserted in the candidate set for this method. But $\mathbf{X}$ would be eliminated by the other method since here *fbound* $= f^*$. However, $\mathbf{X}$ will never be treated by the breadth-first method because of the inequalities in (3). On the other hand, boxes $\mathbf{X}$ with $LB(\mathbf{X}) \leq f^*$ will be inserted either in S or R, independently of the search strategy. Therefore the two methods will treat exactly the same boxes.

3. The parallel methods

The parallel methods are *central* methods where a *master process* communicates with a number of *slave processes*. There is no communication between the slaves. The administration is done by the master which keeps

the candidate set S, and send candidates S_i to the slaves. Each slave perform algorithm (5), and return the result to the master. Thus the master runs algorithm (4) except that *reduce-and-subdivide* is replaced by slave communications. Initially D is subdivided into $p-1$ subsets, $p-1$ being the number of slaves.

The value of *fbound* is known by every slave. If a better value of *fbound* is found by a slave then this is communicated to the other slaves via the master. Results which can be eliminated using *fbound* are not returned to the master.

The parallel *depth-first* algorithm does not always send all results back to the master: If a slave subdivides a box into two then one of them is kept for further treatment by the slave (unless both new intervals can be eliminated), while the other is sent to the master. This of course reduces the amount of communication, since a slave only needs to receive work from the master when it has eliminated or finished its current box.

4. Numerical results

We only present results for two typical test functions. The reader is referred to [3] for further test results. The tests are performed on a Meico system of T800 transputers, where the number of processors used ranges from 1 to 32.

The two functions are

$$f_1(x) = \sum_{j=1}^{10}(\ln(x_j-2))^2 + \sum_{j=1}^{10}(\ln(10-x_j))^2 - (\prod_{j=1}^{10} x_j)^{0.2} \qquad x \in R^{10}$$

$D = [2.001, 9.999]^{10}$

$f_1^* = -45.7784697$, attained in 1 point

$$f_2(x) = \sum_{i=1}^{3}\sum_{j=1}^{5} j\sin((j+1)x_i + j) \qquad x \in R^3$$

$D = [-5, 5]^3$

$f_2^* = -36.0937483$, attained in 1 point

The results are summarized in Tables 1 and 2. In both cases the depth-first method has ideal conditions: f^* is found by the local method. In

both cases the local method (VA13CD of the Harwell Subroutine Library) uses less than one second of computing time. In the tables p is the total number of processors used. ca_p is the calculation time (i.e. essentially the average time the slaves are doing the calculation (9)), and co_p is the communication time (i.e. (the total running time) – ca_p). We also give the maximum number of elements in the candidate set, L_p, and the speed-up, S_p = (total time for 1 processor)/(total time for p processors). All times are measured in seconds.

Under ideal circumstances (i.e. no communication time) we could expect $S_p = (p-1)$, since $(p-1)$ slaves are sharing the work.

	breadth-first method				depth-first method			
p	ca_p	co_p	L_p	S_p	ca_p	co_p	L_p	S_p
1	971.1	–	1645	–	948.0	–	31	–
1+3	313.3	11.2	1641	3.0	313.3	3.1	51	3.0
1+7	134.3	7.7	1650	6.8	134.2	1.9	91	7.0
1+15	62.6	8.9	1661	13.6	62.5	1.9	132	14.7
1+31	30.2	30.6	1648	16.0	30.2	3.7	209	28.0

Table 1. f_1. $p = 1$ corresponds to the sequential method

	breadth-first method				depth-first method			
p	ca_p	co_p	L_p	S_p	ca_p	co_p	L_p	S_p
1	289.3	–	3648	–	237.8	–	11	–
1+3	76.7	40.5	3646	2.5	76.5	3.4	27	2.9
1+7	32.9	59.3	3643	3.1	32.9	2.5	55	6.7
1+15	15.4	69.6	3630	3.4	15.4	2.3	103	13.4
1+31	7.4	87.9	3586	3.0	7.4	7.0	181	16.5

Table 2. f_2. $p = 1$ corresponds to the sequential method

The two methods both parallel rather well on f_1, and the performance of the depth-first algorithm is almost ideal. For f_2, where the amount of work per iteration is much smaller, 32 processors cannot be used quite so efficiently. Further, it is seen from the tables that the depth-first method has several advantages.

First of all the storage needed is drastically smaller. This effect is independent of the fact that *fbound* $= f^*$ initially.

Secondly, as expected the communication time is much smaller in the

depth-first method. For f_2 which is rather "easy" to calculate, the communication time is significant for $p = 4$ and dominating for $p \geq 8$ when the breadth-first method is applied. For the depth-first method, however, the problems only occur for $p = 32$. When solving the first problem a similar comment can be made.

Notice that the depth-first method is fastest, also in the sequential case. This is because the time used to insert in S is smaller (and because the time used by the initial local method is negligible).

However, these advantages of the depth-first method may disappear if the local minimum found initially differs from the global minimum. In that case the depth-first method may have to do more work, since it may be working on intervals with $LB(\mathbf{X}) > f^*$.

5. Conclusion

We conclude that the depth-first method normally requires much less storage than the classical breadth-first method, and it parallels more efficiently. However, the latter advantage may be vasted if the local method applied initially fails to find the global minimum, in which case the total amount of work may increase.

References

1. Hansen, E. R. *Global optimization using interval analysis – the multidimensional case.* Numerische Mathematik **34** (1980), pp. 247–270.
2. Henriksen, T. *Parallelle algoritmer til global optimering.* Masters Thesis. Report NI-E-92-02, Institute for Numerical Analysis, Lyngby, 1992.
3. Henriksen, T. and Madsen, K. *Use of a depth-first strategy in parallel global optimization.* Report NI-92-10, Institute for Numerical Analysis, Lyngby, 1992.
4. Madsen, K. *Parallel algorithms for global optimization.* Report NI-91-07, Institute for Numerical Analysis, Lyngby, 1991.
5. Ratschek, H. and Rokne, J. *New computer methods for global optimization.* Ellis Horwood, 1988.
6. Skelboe, S. *Computation of rational interval functions.* BIT **14** (1974), pp. 87–95.