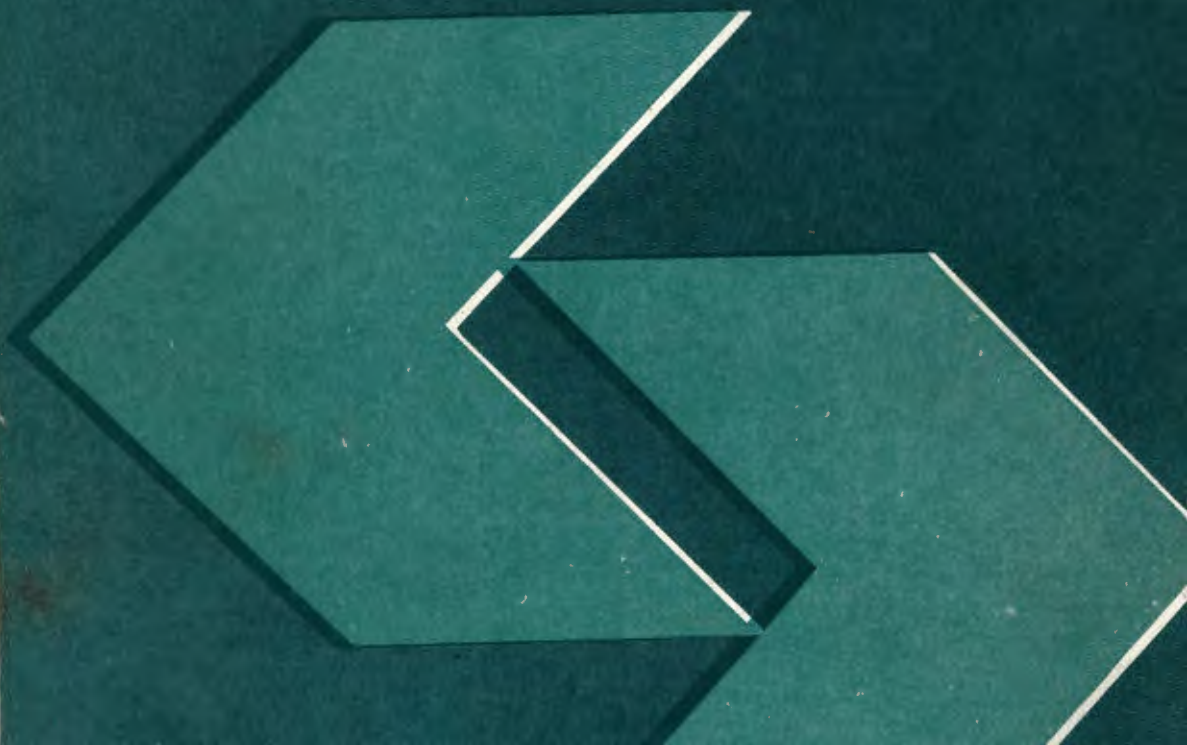


Polska  
Akademia  
Nauk  
Instytut  
Badań  
Systemowych

# Methodology and applications of decision support systems

Proceedings of the 3-rd  
Polish-Finnish Symposium  
Gdańsk-Sobieszewo, September 26-29, 1988

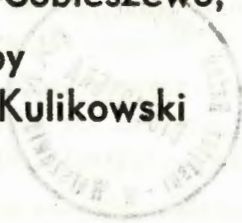
edited by  
Roman Kulikowski





# Methodology and applications of decision support systems

Proceedings of the 3-rd  
Polish-Finnish Symposium  
Gdańsk-Sobieszewo, September 26-29, 1988  
edited by  
Roman Kulikowski



Secretary of the Conference  
dr. Andrzej Stachurski

Wykonano z gotowych oryginałów tekstowych  
dostarczonych przez autorów



41267

ISBN 83-00-02543-X



## Parallel Monte Carlo with application to global optimization

Aimo A. Törn

Åbo Akademi, Data City  
SF-20520 ÅBO, Finland

### Abstract

Different types of parallel machines and existing systems are described and principles for parallelization of algorithms are discussed. Algorithms proposed for global optimization are reviewed and based on this some conclusions regarding parallelization of global optimization algorithms are presented.

### 1 Introduction

With the increasing availability of multiprocessor systems designed for parallel execution of algorithms it seems appropriate to discuss the possible merit in using parallel algorithms in optimization.

One optimization problem that lately has received considerable attention is the so called global optimization problem [Törn and Žilinskas 1988]. The global optimization problem consists in finding the global minimum of a given function  $f(x)$  over some specified region  $A$ . The function  $f(\cdot)$  may have several local minima in  $A$ .

In contrast to local optimization for which the attainment of the local minimum is decidable (gradient equal to zero) no such general criterion exists in global optimization for asserting that the global minimum has been reached.

The only way to obtain information about the function is to make a function evaluation. Normally a procedure for computing  $f(x)$  is available but this may not be the case for the derivatives of  $f$ . Some vague auxiliary information about the smoothness of  $f$  may be available. In addition to determining the global minimum  $f^*$  and  $x^*$  it is of interest to find also other good minima and their locations.

The solution strategy normally consists of global and local stages. In the absence of *a priori* information all parts of the region must be

considered equally critical (the global stage). Of course, no significant part of  $A$  must be neglected, unless one is willing to accept a considerable chance that the global minimum will be missed. When some information is accumulated some parts of the solution set may be deemed more interesting than others and more accurate solutions in these parts are wanted (the local stage).

The effort required to obtain a solution is a function of the smoothness assumptions and the accuracy requirements. Normally it is not known if some smoothness requirement is fulfilled for the problem to be solved and therefore it cannot normally be guaranteed that the global solution is obtained when the computation stops. By assigning liberal smoothness conditions the probability that the optimal solution is obtained will be larger but the computation time will also become longer. To cope with long computation times parallel algorithms could be a possible solution.

Parallel algorithms for different parallel computers are quite different. It is therefore necessary to know the features of these architectures before discussing parallel algorithms. An exposition more detailed than that given below can be found for instance in [Moitra and Iyengar 1987].

## 2 Parallel computers

Parallel computers may be divided into two categories. They are single-instruction multiple-data (SIMD) machines and multiple-instruction multiple-data (MIMD) machines. Ordinary computers with a single CPU could with this terminology be called single-instruction single-data (SISD) machines. MIMD could mean either several SIMD machines or several SISD machines, which is not so important, it is the existence of several CPUs that counts. Instead of SIMD and MIMD the names single CPU (1-CPU) parallel computers and multiple CPU (M-CPU) parallel computers therefore could be used.

### 1.1 Single CPU Parallel Computers

The common feature for these machines is they contain two types of processors; one master processor (CPU) and a large number of simpler slave processors that can work in parallel and essentially perform the same type of task.

Vector processors have already been used some time to speed up processing for some type of problems. The principle is that the same operation is performed simultaneously for a large number of tuples of elements. The speedup is facilitated by the use of vector registers, pipelining, multiple

function units (processors) and chaining. Problems that can be written in form of iterative application of simple operations on a large number of processing elements are suitable for vector processing. Example of vector processing machines are CRAY-1 and CYBER 205 (CDC). Vector processors are also available as add ons for ordinary computers.

A somewhat different single CPU machine is ICL's distributed array processor (DAP). The processors are arranged in a  $64 \times 64$  matrix structure. Each processor can be connected to four neighbour processors; there is also row and column data highways which connect all processors in a given row or column and allow for rapid broadcasting of information across the processing matrix. Each processor has its own local storage and some registers. All the processors perform the same instruction, which is broadcasted from the CPU.

## 1.2 Multiple CPU Parallel Computers

In order for several CPUs to be able to work on the same problem these must be coupled in some way to permit information to be exchanged during the computation. The main principles for this are to have the processors coupled to a common storage or to couple the processors so that they form a network with a distributed storage. The processors can exchange information either by reading from and writing into the same storage positions or by sending messages, depending on the principle adopted.

One class of multiple CPU parallel machines consists of big expensive computers with few CPUs ( $< 10$ ). These machines use the shared storage mechanism. In this class there are several CRAY X-MP machines, IBM 3090 machines and ETA 10 machines (CDC).

Less expensive systems can be constructed by connecting micro computers like workstations to form a parallel computer. Such an environment consisting of Sun-3 workstations connected on an ethernet exists at the University of Colorado. The ability to use the network of workstations for distributed concurrent processing is based on the Sun version of the Berkeley Unix 4.2 operating system, which each workstation runs. A distributed processing tool, DPUP, provides two models of concurrent computation. The first is a master-slave model where all processes are linked to one master in a star arrangement and all communication is through the master. The second is a broadcast model where each process is an equal member of a ring of processes and can send messages to all of the processes at once. DPUP provides several basic concurrency capabilities including the creation and termination of remote processes and various means to send and receive messages.

A third class consists of inexpensive micro processors that are connected to become nodes in a processor network. The input/output unit and the secondary storage used is normally supplied by an ordinary micro computer. An example of such processors are Inmos' transputers, especially designed to be nodes in a distributed processor network. The transputers are rather powerful CPUs (IMS T414:  $\approx 10$  Mips, IMS T800:  $\approx 2$  Mega Flops) with its own local memory. Each transputer may be connected to four other transputers.

The simple construction admits to construct large systems with a considerable computing power (super computer) at a rather low cost. Because of this we believe that the utilization of these machines will probably show to be highly rewarding. The concept of shared storage may be used and it can be implemented without having a physically shared storage [Ahuja *et al* 1986].

## 2 Parallel Algorithms

The performance of parallel algorithms is often measured by using the characteristic called *speedup*. Let the time needed to solve a problem by a parallel algorithm on  $m$  processors be  $t_m$  and let the time required to solve the same problem using a non-parallel algorithm on the same system be  $t_1$ . The speedup  $S_m$  is defined as the ratio

$$S_m = t_1/t_m.$$

If the speedup is proportional to the number of processors  $m$  used it is said that the speedup is linear. The expression for the speedup could also be interpreted as the *effective* number of processors  $m_e$  in the system.

The *efficiency* of an implementation of an algorithm on a multiprocessor machine may be defined as

$$E_m = S_m/m.$$

In order to obtain a good efficiency one should therefore keep all the processors busy working productively until the problem is solved, and in order to obtain good speedup one should try to use as many processors working as possible.

The *effectiveness*  $F_m$  [Schendel 1984] is defined as

$$F_m = S_m/(mt_m) = E_m \cdot S_m/t_1,$$

where  $mt_m$  measures the 'cost' of the parallel algorithm.  $F_m$  is thus a measure of both speedup and efficiency.



## 2.1 Principles for Parallelization

The central problem of parallel programming is the question of how to obtain program units that can be processed in parallel. There are at least two issues that must be addressed when discussing parallelization. The first is the degree of parallelism (speedup and efficiency) and the second is the programming effort needed to obtain a correct program.

Based on the presentations in [Ahuja *et al* 1986; Hey and Pritchard 1987] we recognize the following classes of parallelism: *Monte Carlo Parallelism* (processor farm), *Geometric Parallelism*, *Task Parallelism* and *Algorithmic Parallelism*.

The Monte Carlo method consists of performing a number of experiments which can be seen as drawing independent samples from an unknown distribution. Using these the characteristic of interest, whose value may be deterministic is estimated. The accuracy of the estimate is increasing with the number of experiments.

For problems solvable by a Monte Carlo method there is a natural possible parallelism in the sampling process. This sampling could be performed in parallel on  $m$  processors by running the same program for different random numbers. This parallelism that may be called Monte Carlo parallelism is therefore very easy to implement. The speedup is linear and the efficiency maximal because no interaction between processors is needed.

Geometric parallelism is achieved by partitioning the problem into  $m$  subproblems of the same kind as the original one for instance by dividing the decision space into  $m$  parts. Also in this approach the programming is reduced to developing a single program and the workload on different processors is balanced. Of course one would expect to encounter some border problems whose solution will require interaction between neighbouring processors. This could lead to some loss in efficiency. It is interesting to note that Monte Carlo parallelism can be regarded as that special case of geometric parallelism for which the  $m$  parts are identical to the original problem.

For task parallelism the problem is partitioned into a number of different tasks classes. The tasks to be performed are administrated by a cocordinator. When a processor becomes idle it will be assigned a new task by the coordinator. The speedup and efficiency is dependent on the number of tasks that can be performed in parallel. The efficiency may be improved by decreasing the number of processors used. The workload is naturally balanced.

For algorithmic parallelism the steps  $i = \overline{1, k}$  in the program are realized as  $m_i$  parallel processes, where  $m_i$  is determined by the logic of the

algorithm. Each of the  $m_i$  logical processes can run on a separate processor. It is not an easy task in this case to achieve efficiency because the number of processors needed in the different steps may differ essentially. Also it is to be expected that the different logical processes in the steps  $i$  and  $i + 1$  will correspond to completely different algorithms which means context switching and difficult programming.

## 2.2 Parallel Global Optimization

For a more general discussion including local optimization see [Sutti 1983a,b; Schnabel 1984,1987; Iootsma and Ragsdell 1988].

Parallelism is easily achieved in global optimization. By partitioning the optimization region  $A$  in  $m$  parts geometric parallelism is obtained.

For probabilistic methods both geometric and Monte Carlo parallelism can be utilized. Geometric parallelism would be more efficient because of the stratified sampling, i.e., the possibility that some region is ignored is smaller for the more uniform distribution of trial points of the stratified sampling and therefore fewer points need to be sampled [Törn 1974].

However, the partition may lead to a need for neighbouring processors to communicate. For instance, if local search is a component in the global optimization method used a working point may want to enter the region of a neighbouring processor. One solution would be to "transport" this point to the processor in question which then could continue the local search. This could lead to big differences in processor workload. Another solution which do not have this drawback is to use the partition only for sampling trial points and let the search region be the whole  $A$  for each processor. If found advantageous new local minima found could be communicated to all processors. In this case the communication need could be expected to be smaller and the workload balanced.

## 3 Applications

One of the earliest contributions to parallel global optimization is [McKeown 1980]. Based on a discussion of Törn's algorithm [Törn 1978] a general approach to designing parallel algorithms for multiple CPU systems utilizing global memory is proposed. In order to achieve the highest level of parallelism possible he suggests modifying the complete sequential algorithm so that it may be applied as a whole as one of many identical parallel tasks, each of which when run alone would solve the problem. He calls such an algorithm a *holistic* algorithm. The proposed holistic Törn algorithm uses the global memory to store the working points (initially  $N_0$

in number). Each task improves working points and periodically transfers them to private memory for clustering and reduction. Some measures for avoiding unwanted interference between tasks such as flagging points that are being improved are discussed. However, such interference problems would not exist if Törn's original algorithm as such were used as the holistic algorithm, initializing the random number generator differently for each of the  $m$  parallel task and using  $N_0/m$  initial working points for each task, i.e., if Monte Carlo parallelism were used. McKeown also suggests a holistic version of the Price algorithm [Price 1978]. Some results of simulating the effect of various parameters in the algorithm on such characteristics as the proportion of time wasted in queuing for access to global memory is reported.

Dixon and Patel [Dixon and Patel 1981] discuss the parallelization of the algorithm of Price [Price 1981] for single CPU parallel computer (ICL's DAP), the algorithms of Törn [Törn 1978], and Boender *et al* [Boender *et al* 1982] for a multiple CPU machine with global memory. The algorithms suggested are based on algorithmic parallelism. For Törn's algorithm a task parallelism approach, where some processors are dedicated to perform tasks from the same task class is also suggested. The tasks considered are: sampling points, improving points and reducing points. No experiments were performed.

In [Schnabel 1984] a conceptual parallel algorithm for global optimization based on algorithmic and geometric parallelism is suggested. The algorithm basically consists of adaptively partitioning the variable space into subregions likely to contain one local minimizer each (parallel sampling, throwing away high points and clustering), and then simultaneously finding these local minimizers using separate processors, recurring this procedure as necessary. Additional efficiency is promoted by the early termination of subregions where the function is high.

In [Sutti 1984] a parallelization of Price's algorithm for a multiple CPU computer based on algorithmic parallelism is suggested. It is reported that numerical experiments on the Neptune system at Loughborough University were performed on parallel versions of Price's and Törn's algorithms. No numerical results are given.

Evtushenko [Evtushenko 1985] in his description of covering methods for global optimization suggests the use of multiprocessor computers. The operating efficiency of the method depends on the sequence of coordinate-wise coverings of the parallelepiped. By running one processor for each coordinate and exchanging information between the processors the efficiency could be improved. He also suggests another possible variant: the partition of the parallelepiped into  $s$  parts,  $s$  being the number of processors, i.e., geometric parallelization.



### 3.1 Single CPU Applications

Two implementations of Price's algorithm on the ICL DAP at Queen Mary College, University of London was applied to all except H3 of the standard test problems and additionally to the six-hump camel back function and a Shubert function with 18 global minima [Ducksbury 1986]. The initial parallel algorithm which was a direct parallelization of Price's algorithm was much slower than the sequential algorithm! The explanation is that the three sections of the algorithm that could not be parallelized were the main time consumers. The algorithm was then modified to allow for more parallelism. This version showed a speedup ranging from 3.2 to 68 over the problems. The efficiency is still rather low because 4096 processors were used. The experiment revealed that the parallelism in the algorithm must be very closely matched onto the available hardware, in order to take advantage of its computing power

Large scale ( $n = 100 - 500$ ) quadratic (with negatively defined matrix) global optimization problems solved on vector computers is considered in [Pardalos 1986]. The optimizations were carried out on the machines Cray 1S and Cray X-MP/48. With only 4 exceptions (out of a total 240) test problems, at least one solution was found to each problem. No advantages of the multiple CPU capability of the Cray X-MP/48 was taken. The CPU time for the problems was found to be well approximated by  $(1.5/10^4)n^4$  sec.

### 3.2 Multiple CPU Applications

In [Byrd *et al* 1986; Schnabel 1987] a concurrent variant of Timmers algorithm is considered. The test environment consisted of a network of four or eight Sun-3 workstations. The master-slave model of DPUP was used (see Sec. 1.2). The parallel algorithms are based on algorithmic parallelism the first part using geometric parallelism (sampling and choice of start points) and the second part task parallelism (local minimizations) [Byrd *et al* 1986].

The test problems considered were the standard test problems. From their results they can simulate the speedup that would be obtained on these problems using any number of processors when only the cost of function evaluations is being considered. With a sample size of 200 points per iteration, the speedup with 8 processors was between 3.5 and 6.1 for the seven test problems, and for the number of processors 200 or more 4.8 and 10.0 respectively. For a larger sample size; 1,000 points, the speedup figures were 6.0-7.1 (8 processors) and 17.9-27.0 (1,000 processors).

The results illustrate that it is rather easy to make good use of a relatively small number of processors for expensive function evaluations using this algorithm, but that the speedup for larger number of processors



may be limited due to the small number of local minimizations in the second part of the algorithm.

In order to make use of more processors when performing local minimizations Schnabel proposed to use several processors in the local minimization step so that function evaluations and finite difference gradient evaluations could be made in parallel. The speedup for the H3 problem for (6, 12, 24, 1000) processors improved from (5.4, 8.7, 12.9, 23.9) to (5.8, 10.5, 17.1, 44.0) and (5.8, 11.3, 20.5, 76.3) for two different parallel local minimization strategies. The improvement is essential, but for a large number of processors the speedup gain declines. In order to further improve the efficiency it is suggested to divide the problem into task classes (sampling, minimization). By dedicating the right number of processors for performing tasks from these task classes a more efficient utilization of the processors overall would be achieved.

At Abo Akademi (Finland) some experiments were performed on a 16 IMS T414 transputer system (Hathi 1) [Walldén 1987]. Geometric parallelization was used dividing  $A$  into 8 equal sized parts in which crude sampling was performed using 8 processors. The program was written in OCCAM, a language specially designed for parallel programming. In order to introduce some communication the processes were connected in a tree structure each process reporting when a new better value was found. In this way the top process all the time has up-to-date information about the best point discovered. This best point value was also communicated down the tree in order to avoid unnecessary reporting. The speedup approaches 8 for expensive functions as expected. When compared to the parallel algorithm run on a single transputer the speedup was as high as 14 because of the time-slicing overhead when the processes are running in parallel on the timesharing system of one transputer. A more serious effort to parallel global optimization will be undertaken on Hathi 2, a 100 IMS T800 transputer system installed in 1988.

Price describes a transputer version of his algorithm [Price 1978, 1983] in [Price 1987]. Each transputer is working on its own (randomly chosen) set of points, trying to replace the worst point by a better. When such a point is found the result is forwarded to a coordinator transputer for acceptance. The coordinator transputer also acts as the storage for the whole point set and initiates local searches when some conditions are fulfilled. Simulations with an OCCAM implementation suggest that the speedup should be linear provided that communication overheads are relatively insignificant. It is also mentioned that some preliminary experiments run on a prototype computer with five transputers achieved the expected fourfold reduction in running time.

#### 4 Conclusions

The speedup and efficiency obtained in parallel applications of global optimization has in many cases fallen short of expectations. In some cases the time to obtain a solution has even been longer for a parallel algorithm than for a corresponding sequential one. Partly this is due to the novelty of the subject, and to the fact that programming parallel algorithms is more difficult than programming sequential ones. However, a more fundamental difficulty is the limited parallelism achievable in algorithmic parallelization of certain sequential algorithms. Because global optimization is suitable for Monte Carlo and geometrical parallelization, which does not have the drawbacks of algorithmic parallelization, methods based on these concepts as well as algorithms based on task parallelism seems worth future investigation.

The speedup possible by using parallel computers is especially interesting for the covering type methods because successful application here will considerably extend the number of solvable global optimization problems.

#### References

- [Ahuja et al 1986] S. Ahuja, N. Carriero and D. Gelernter, *Linda and friends*, Computer 19, 26-34.
- [Boender et al 1982] G. Boender, A. Rinnooy Kan, L. Stougie and G. Timmer, *A stochastic method for global optimization*, Mathematical Programming 22, 125-140.
- [Byrd et al 1986] R.H. Byrd, C. Dert, A.H.G. Rinnooy Kan and R.B. Schnabel, *Concurrent stochastic methods for global optimization*, Tech. Rept. CU-CS-338-86, Department of Computer Science, University of Colorado, Boulder, CO, 40.
- [Dert 1986] C.L. Dert, *A parallel algorithm for global optimization*, Masters thesis, Econometric Institute, Erasmus University, The Netherlands.
- [Dixon and Patel 1981] L.C.W. Dixon and K.D. Patel, *The place of parallel computation in numerical optimization II; the multiextremal global optimization problem*, The Hatfield Polytechnic, NOC 119, 11 p.
- [Ducksbury 1986] P.G. Ducksbury, *Parallel array processing*, (Ellis Horwood, Chichester) 123 p.
- [Evtushenko 1985] Yo.G. Evtushenko, *Numerical optimization techniques*, (Optimization Software, Inc., New York), 561 p.



- [Hey and Pritchard 1987] A.J.G. Hey and D.J. Pritchard, *Parallelism in scientific programming and its efficient implementation on transputer arrays*, Technical Report, Dept. of Electr. and Computer Science, University of Southampton, 70 p.
- [Lootsma and Ragsdell 1988] F.A. Lootsma and K.M. Ragsdell, *State-of-the-art in parallel nonlinear optimization*, *Parallel Computing* 6, 133-155.
- [McKeown 1980] J.J. McKeown, *Aspects of parallel computations in numerical optimization*, in: F. Arcetti and M. Cugiani (eds.), *Numerical techniques for stochastic systems*, 297-327.
- [Moitra and Iyengar 1987] A. Moitra and S.S. Iyengar, *Parallel algorithms for some computational problems*, In: M.C. Yovits (ed.), *Advances in computers* 26 (academic Press, Boston), 93-153.
- [Pardalos 1986] P.M. Pardalos, *Aspects of parallel computation in global optimization*, *Proc. of the Annual Allerton Conf. on Communication, Control and Computing* 24, 812-821.
- [Price 1978] W.L. Price, *A controlled random search procedure for global optimization*, in: Dixon, L.C.W. and G.P. Szegö, (eds.), *Towards Global Optimization 2* (North-Holland, Amsterdam) 71-84.
- [Price 1981] W.L. Price, *A new version of the controlled random search procedure for global optimization*, Technical Report, Engineering Dept., Uni. of Leicester.
- [Price 1983] W.L. Price, *Global optimization by controlled random search*, *JOTA* 40, 333-348.
- [Price 1987] W.L. Price, *Global optimization algorithms for a CAD workstation*, *JOTA* 55, 133-146.
- [Schendel 1984] U. Schendel, *Introduction to numerical methods for parallel computers*, (Ellis Horwood, Chichester) 151p.
- [Schnabel 1984] R.B. Schnabel, *Parallel computing in optimization*, In: K. Schittkowski (Ed), *NATO ASI Series Vol. F15, Computational Mathematical Programming* (Springer-Verlag, Berlin), 357-381.
- [Schnabel 1987] R.B. Schnabel, *Concurrent function evaluations in local and global optimization*, *Computer Meth. in Appl. Mech. and Engineering* 64, 537-552.
- [Sutti 1983a] C. Sutti, *Nongradient minimization methods for parallel processing computers*, part 1, *JOTA* 39, 465-474.
- [Sutti 1983b] C. Sutti, *Nongradient minimization methods for parallel processing computers*, part 2, *JOTA* 39, 475-488.
- [Sutti 1984] C. Sutti, *Local and global optimization by parallel algorithms for MIMD systems*, *Annals of Operations Research* 1, 151-164.
- [Törn 1974] A.A. Törn, *Global optimization as a combination of global and local search*, Åbo Akademi, HIIÅA A:13, Finland, 65p.

- [Törn 1978] A.A. Törn, *A search clustering approach to global optimization*, in: Dixon, L.C.W. and G.P. Szegö, (eds.), *Towards Global Optimization 2* (North-Holland, Amsterdam).
- [Törn and Žilinskas 1988] A.A. Törn and A. Žilinskas, *Global optimization*, to appear, 250 p.
- [Walldén 1987] M. Walldén, *Performance of a distributed algorithm*, Technical Report B 5, Abo Akademi (Finland) Press, Dept. Comp. Sc., 31 p.
- [Walster et al 1985] G.W. Walster, E.R. Hansen and S. Sengupta, *Test results for a global optimization algorithm*, in: P.T. Boggs, R.H. Byrd and R.B. Schnabel (eds.), *Numerical optimization 1984*, SIAM, Philadelphia 1985, 272-287.





IBS

41267