Parallel biased search for combinatorial optimization: genetic algorithms and TABU

Roberto Battiti* and Giampietro Tecchiolli† consider some relatively general techniques for solving combinatorial optimization problems, ranging from concurrent execution of independent searches to a fully interacting 'population' of tentative solutions

Combinatorial optimization problems arise in different fields and require computing resources that grow very rapidly with the problem dimension. Therefore the use of massively parallel architectures represents an opportunity to be considered, characterized by very large speed-ups for significant applications. In this paper we consider some relatively general techniques that are paradigmatic of different parallel approaches, ranging from the concurrent execution of independent searches to a fully interacting 'population' of candidate solutions. In particular, we briefly summarize the TABU and GA algorithms, discuss their parallel implementation and present some experimental results on two benchmark problems: QAP and the N-k model. A new 'reactive' TABU scheme based on the 'open hashing' technique is also presented.

combinatorial optimization **parallel computing** **genetic algorithms** **TABU**

We consider some general parallel schemes for solving combinatorial optimization problems, where the domain of the function to be minimized consists of a finite set of possible configurations, although their number grows rapidly (e.g. exponentially) with the characteristic size N of many significant applications. The range of the function is assumed to be in the set of non-negative integers.

The theory of computational complexity makes a machine-independent distinction between problems that can be solved with computing times that are polynomial in the problem size N, and intractable problems. In particular, for the class of NP-hard optimization problems, there is little hope that algorithms solving them in polynomial time will ever be found (see Reference 1 for the theory).

Many interesting problems are of this kind. Because the main topic of this paper is parallel implementation, we present here only a short qualitative explanation. Generally applicable optimization strategies are based on iterative search techniques, in which an initial (possibly randomly generated) candidate solution is progressively improved by a set of elementary operators ('local movements'). The solution traces a path in the problem configuration space. A sensible strategy is to arrange the movements so that the path reaches lower and lower values of the function to be minimized. In particular, the 'greedy' steepest descent strategy achieves the fastest decrease by choosing the movement that causes the maximum decrease at each point. The descent is stopped when a local minimum is encountered. In many cases finding the 'nearest' local minimum is relatively inexpensive, but the number of local minima grows more than polynomially and, furthermore, the relationship between the various local minima and the global minimum is complex, so that the problem cannot be solved exactly in polynomial time.

To help with intuition it is customary to talk about the 'fitness surface' of a problem, where the height of a point s is given by the value of the fitness function $f(s)$, the function to be optimized. The complexity of the problem is pictured as the roughness of the surface, with many peaks indicating a complex structure. Note that maximization becomes (trivially) minimization by multiplying the function by -1. In spite of the theoretically motivated negative result about the asymptotic behaviour of solution times, limiting the consideration to local minima

*Dipartimento di Matematica, Università di Trento, 38050 Povo (Trento), Italy and INFN Gruppo Collegato di Trento
†Istituto per la Ricerca Scientifica e Tecnologica, 38050 Povo (Trento), Italy and INFN Gruppo Collegato di Trento
Paper received: 18 June 1992

0141-9331/92/070351-17

('repeated local minimum search' - RLMS) may produce a drastic decrease in the number of configurations to consider. RLMS consists of generating a random initial point, using steepest descent to arrive at a local minimum, and repeating this procedure until the desired solution is found.

The various optimization algorithms differ by the mechanisms with which 'bias' toward points with lower function values and 'escape' from local minima are realized. The 'bias' can either be explicitly executed through a search of local minima, or, implicitly, through a drift of candidate solutions. In some cases the 'bias' and 'escape' behaviour are the emergent properties of complex interactions between candidate solutions. In this paper we review a method based on a local escape with cycle-avoidance techniques (TABU) and a method based on combining building blocks from a pool of candidate solutions (genetic algorithms - GA). These methods are advantageous with respect to RLMS only if they can exploit regularities in the objective function. On the other hand, it is known that no 'blind search' algorithms (whose only information sources are the function values) can expect to do better than random search on a generic problem[2]. For this reason, we cannot expect that a black-box algorithm will efficiently solve all the possible optimization problems. The probability of finding the desired optimal solution near the current sub-optimal point (in the TABU case) or as a combination of parts of sub-optimal points (GA) must be higher than that of finding it by randomly restarting the process and progressing toward a new and uncorrelated local minimum.

The cases of TABU and GA are paradigmatic because they lead the way to different parallel implementations, in the first case based on the independent execution of search processes, and in the second based on a population of interacting tentative solutions. We do not consider schemes where parallelization is applied to evaluate the function, both because they depend crucially on the specific problems and because these schemes tend to possess only a limited amount of concurrency.

In the following sections we first summarize the main features of the TABU and GA algorithms, then we describe two test problems (the quadratic assignment problem (QAP) and N-k model) and the statistical tools used for the parallelization. Finally, we present some possible parallel schemes with some indicative experimental results.

TABU SEARCH

The TABU search meta-strategy is built upon a descent mechanism of a search process. The descent biases the search toward points with low function values, while special features (the TABU list(s)) are added to avoid being trapped in local minima, without renouncing the stepwise movement of the candidate solution executed by a set of elementary operations. The seminal idea of the method is introduced in Reference 3, while a detailed description is contained in References 4 and 5.

The TABU scheme strikes a balance between 'exploitation' (i.e., intensification of the search in the neighbourhood of the current sub-optimal solution) and 'exploration' (i.e., the diversification of the search to hitherto unexplored regions of the configuration space). If the first component is missing the search becomes an iterated random sampling, if the second is missing the process may be trapped in a sub-optimal region.

The main problem with trying to leave a local minimum and remain in the nearby region is that the point may cycle and return again to the same point soon after leaving it.

KEEP MOVING, AVOID CYCLING

To sketch the TABU cycle avoidance mechanism, let us consider a set of elementary moves $M_1, \ldots M_n$ applicable to a current configuration s (for example, in the QAP problem described below, a move consists of exchanging the locations assigned to two units). The neighbourhood $\Omega(s)$ is the set of all configurations that can be reached from s by applying one of the elementary moves.

The basic local improvement technique consists of evaluating a subset $\Theta(s) \subseteq \Omega(s)$ of points in the neighbourhood and moving to the best of them, say $s' = M_a(s)$. If the entire neighbourhood is evaluated (i.e., $\Theta(s) = \Omega(s)$) this becomes a 'least ascent' technique. Note that the movement is executed even if the value of the function increases ('keep moving').

Now, if s is a local minimum, $f(M_a(s))$ will be greater than $f(s)$ and then there is substantial probability that $M_b(M_a(s)) = s$, therefore creating a cycle of length 2. The 2-cycle is avoided if the inverse movement M_b is forbidden (i.e., it becomes 'taboo') for a given number of iterations. In general, the basic data structure of TABU is a finite-size list of forbidden moves. In certain implementations, the list contains attributes, which implicitly impart a taboo classification to moves that contain them. The structure is logically a circular list, with a first-in first-out mechanism, so that the latest forbidden move replaces the oldest one. The finite memory mechanism is crucial because a movement in the TABU list may become necessary to reach the optimal point in subsequent phases of the search. While the TABU list mechanism does not guarantee the absence of cycles, their occurrence is rare if the list size is sufficient. After a point has moved away from a local minimum through a long sequence of moves, even if the inverse of the first moves are allowed again, returning to the initial point becomes very unlikely.

The TABU literature presents a wealth of prescriptions for the suitable TABU list size and for the appropriate balance of the short term (i.e., the TABU list) and the long term memory components, in order to obtain the desired search exploration (see for example References 4 and 5, and the contained bibliography). The skeleton for an optimization scheme based on TABU is illustrated in Figure 1.

It is crucial that the cycle avoidance part does not impose a high computational burden on the global algorithm.

TABU with hashing for cycle detection

General purpose optimization algorithms should not require extensive tuning of a number of critical parameters for their success. While the original TABU scheme is based on fixed length lists, there have been various proposals to make the algorithm more robust, by dynamically adapting the search strategy to the history of the process (long term

```
int time=0;                                  /* iteration counter                                  */
int current_value, best_so_far;              /* function values during search                      */
int sub_optimum;                             /* termination value (for benchmarking only!)         */

int tabu_minimize()
{
        time=0;
        initialization();                    /* memory allocation, random generation of            */
                                             /* initial configuration                              */
        while(best_so_far > sub_optimum)
                {
                  evaluate_neighborhood();   /* for all possible exchanges of units r and s,       */
                                             /* calculate move_value[r][s], such that function     */
                                             /* value after exchange = current_value-move_value[r][s]*/

                  choose_movement();         /* take the best exchange that is not TABU or that    */
                                             /* reaches the aspiration level                       */

                  time++;
                  bookkeeping();             /* update current_value, best_so_far, collect statistics*/

                  make_tabu(r_chosen);       /* record the last occupation time for the chosen units */
                                             /*  in the locations occupied before the exchange     */
                }
}

int r_chosen,s_chosen;                       /* indices of  units exchanged in the chosen movement */

void   choose_movement()                     /* take best exchange of units that is not TABU or that */
                                             /* reaches the aspiration level                       */
{

int maxdecrease;                             /* decrease in function value                         */
int r,s;                                     /* indices of units exchanged                         */

        maxdecrease= -INFINITY;

        for(r=0;r<N-1;r++)
                {
                  for(s=r+1;s<N;s++)                 /*  for all exchanges, N=number of units     */
                        if(move_value[r][s] > maxdecrease)
                                {
                                  if(is_not_tabu(r,s) || aspiration_level_reached(r,s))
                                        {
                                          maxdecrease = move_value[r][s];
                                          r_chosen    = r;
                                          s_chosen    = s;
                                        }
                                }
                }
}

int    aspiration_level_reached(r,s)         /* aspiration OK if the new point has a function value */
                                             /* lower than the best ever found                     */
int r,s;
{
        return((current_value-move_value[r][s]) < best_so_far);
}

int is_not_tabu(r,s)

int     r,s;
{
        /* returns 1 if exchange is not TABU,0  otherwise. An exchange is TABU if it places      */
        /* both units to locations that they had occupied within the list_size most recent      */
        /* iterations, see text                                                                 */
}
```

*Figure 1. Skeleton of TABU search program for the quadratic assignment problem illustrated in the text. The description is in C language ('var++' is the assignment 'var = var +1', 'var1 *= var2' means 'var1 = var1 * var2', '||' is the logical OR operator, '!=' means 'different from'), with comments (/* ... */)*

memory) or by randomly changing the length of the TABU list, to further decrease the probability of long-lasting cycles (see for example Reference 6).

Here we present a new proposal, which does not need any parameter tuning by the user, has a negligible overhead during the search process (the added operations are executed in almost constant time, independently of the problem size) and, finally has been shown to converge efficiently on various benchmark problems.

The proposal is that of detecting potential cycles by checking for the repetition of function values. A cycle of length n implies that a given configuration is visited again after n iterations, producing the same function value. The inverse implication is not always true (the same function value may happen by chance, produced by a different configuration), but we assume that the probability of its happening by chance (in a relatively short period) is small. If this is not the case, the strategy can easily be modified:

the detection of cycles will be based on comparing the frequencies of repetitions with a threshold derived from estimates of the 'spontaneous' frequencies, i.e., of the probabilities for casual repetitions.

Let us assume that our elementary operators M_i are such that a TABU list of length L_T avoids cycles of a length up to L_T. For example, this is assured if the M_i flip individual bits in a binary string. In reality, our algorithm requires only that the minimum length of possible cycles be approximately proportional to L_T.

Now, the proposal for dynamically adapting the TABU list size to the problem is the following. Start with a small size (0 is acceptable) and, after each iteration, check for the possible presence of cycles by observing the repetitions of function values. If a possible cycle of length L_c is detected, increase the list size so that it becomes equal to L_c (this is a simple 'reaction' mechanism; more complex schemes are currently under investigation). If only increases are allowed, the list size may become excessive in different regions of the search space, causing search inefficiencies. Therefore, a second mechanism with slower dynamics gradually reduces the length in an exponential way. The actual list size will be produced by a balance between the fast 'reaction mechanism', immediately forcing the search out of cycles, and the slow reduction process.

In order to cause a negligible overhead, the detection mechanism is executed by the open hashing technique[7]. The couples [value, iteration number] are stored in memory at an address ('bucket') that is obtained by a 'scattering' function of the values, so that both lookup and insert operations are executed in a very small number of machine cycles (approximately constant, $O(1)$). Collisions (values with the same address) are taken care of with a list of elements for each bucket. The basic scheme of the TABU with hashing technique is illustrated in Figure 2. In Table 1 we compare the results of Reference 6 (TABU with randomly varying list size), the RLMS search and the proposed TABU with hashing technique. Averages for our tests are calculated on 10 successful runs. In approximately 10% of the tests the search does not converge, because casual repetitions produce excessive list sizes. The refinements to deal with this problem are beyond the scope of this paper, which is dedicated to the parallel implementation. The results indicate that i) the RLMS technique achieves average solution times comparable (although 3–4 times larger) to those of TABU (it is plausible that most of the advantage with respect to a repeated random extraction is caused by the search for local minima, with TABU providing added efficiency through a local search intensification); ii) the TABU with hashing scheme attains results that are comparable with those of TABU with a randomly varying list size. In addition, the hashing scheme 'learns' a suitable list size during the initial part of the search (this explains its inefficiency for very small problem sizes) and keeps it 'adjusted' to the local properties of the energy surface, with no user intervention.

Table 1. Comparison of TABU with randomly changing list size, repeated local minima search and TABU with hashing, on the QAP problem of different sizes *N*. The values listed are the number of steps executed

N	TABU (from Taillard)	RLMS	TABU with hashing
10	137	297	200
15	2168	6880	1871
17	5020	15193	5632
20	34279	97120	16599
25	80280	N.A.	48458
30	146315	N.A.	92229
35	448514*	N.A.	311020

*This result required the use of a second aspiration function[5]

A digression about simulated annealing

The difficulty in escaping from a deep local minimum without returning to it can be responsible for the slow progress rate in an algorithm like simulated annealing[8], in which the probability of accepting a move that increases the function by Δf is proportional to $\exp\{-(\Delta f/T)\}$. Let us consider the situation in which the current search configuration is in a suboptimal region separated from the optimal point by a high barrier, requiring a long chain of local movements in order to be passed. If the 'temperature' parameter T is much less than the height of the barrier, the occurrence of a suitable chain of upward movements has an exponentially decreasing probability. In practice, for the available amount of computing time, the system will jump a little upward, but continue to fall back to the local minimum. Practical implementations of the SA algorithm resort to fast cooling schedules and there is evidence that these schemes are a form of 'multiscale' optimization[9], where the stochastic component can be substituted with a deterministic rule and the solution point is first 'trapped' in a local valley at a large scale and then adjusted to finer and finer details when the temperature is lowered. A proposal for embedding a deterministic local search technique into stochastic algorithms is presented in Reference 10, where SA is based on large steps between different local minima.

GENETIC ALGORITHMS

Genetic algorithms[11] try to optimize a function by mimicking natural selection mechanisms, where the competition among individuals increases the average fitness of a population. GAs operate on a population of strings, representing the encoding of points in the domain of f, and use probabilistic genetic-like operators to produce new populations, where the fitter individuals tend to have larger offspring. Because of the three main ingredients (see, for example, Reference 2): i) the encoding of the search space, ii) the dynamics induced by genetic-like operators and iii) the intrinsic probabilistic approach, a GA can efficiently scan wide areas of the search space concentrating on the more promising regions and move away from the low-fitness ones without using any explicit memory mechanism. Both for GA and TABU the right combination of encoding and elementary operators is critical for the efficiency and efficacy of the search. The operators produce new candidates starting from the current points. Their action must not completely destroy the information content of the current configuration (otherwise the search becomes random) and must permit the efficient exploration of the entire search space.

```
int tabu_with_hashing()

{
        time=0;

        initialization();
        while(best_so_far > sub_optimum)
                {
                  evaluate_neighborhood();
                  choose_movement();
                  time++;
                  bookkeeping()
                  make_tabu(r_chosen,s_chosen);
                  check_for_possible_cycle(current_value);
                }
}

#define CYCLE_MAX 50

void check_for_possible_cycle(value)            /* update hashing table and check for a cycle */

int value;

{
struct nlist    *np;                            /* pointer to structure containing [value, last_time]*/
int             length;

        if((np=lookup(value)) != NULL)                  /* if value is foud in Hashing table    */
                {
                  length = time - np->last_time;        /* length of possible cycle             */
                  np->last_time= time;                  /* rewrite last_time at which value     */
                                                        /* was encountered                      */
                  if(length <= CYCLE_MAX)
                                                        /* value repetion after a short interval */
                                                        /* react immediately to eliminate cycles */
                        {
                          if(list_size < length )
                                {
                                  list_size = length    /* increase TABU list size              */
                                  last_change_of_list_size = time;
                                }
                        }
                }
        else
                {                                       /* if value not found in hashing table  */
                  install(value,time);
                  if((time-last_change_of_list_size) > (2*CYLE_MAX))
                        {
                          list_size *= .95;             /* decrease TABU list size */
                          last_change_of_list_size = time;
                        }
                }

}

struct nlist *lookup(value)
int value;
{
        /* if value is found (in approximately constant time), returns a pointer to structure   */
        /* containing [value,last_time], otherwise it returns a NULL pointer                    */
}

void install(value, last_time)
int value, last_time;
{
        /* find bucket with hashing function and install the couple [value, last_time]          */
}
```

Figure 2. Skeleton of program for the TABU with hashing algorithm. The hashing procedures **lookup** *() and* **install** *() are implemented with the standard open hashing technique. The description is in C language with comments*

A concurrently evolving population of 'searchers'

In this paper we concentrate only on a relatively general and well performing version of GAs, usually named 'GA with stochastic selection and replacement, uniform crossover and mutation' (for other variants see for example References 12 and 13). We refer to it by the term 'classical GA'. Different GAs are determined by a large variety of genetic operators (for a general discussion see Reference 14, for a study of selection operators see Reference 15, for crossover operators see Reference 16, and for the role of mutation see Reference 17).

Let S be an alphabet of symbols such that $2 \leqslant |S| < +\infty$, D the search space and e an encoding function such that:

$$e : S^l \rightarrow D (l \geqslant 2,\ l \in \mathcal{N}) \tag{1}$$

S^l is the set of strings of length l composed of the symbols of S. Let us denote with s_j the jth symbol of $s \in S^l (0 \leqslant j < l)$. In the language of GAs S is the set of alleles, S^l is the set of

chromosomes or genotypes or individuals, D is the set of phenotypes, s_j is a gene and j is the locus of s_j in s. Let the optimization task consist of finding the maximum value of an objective function $h : D \to \mathcal{N}$. The fitness function is the composite map $h \circ e$:

$$f \equiv h \circ e : S^l \to \mathcal{N} \tag{2}$$

$$f : S \xrightarrow{e} D \xrightarrow{h} \mathcal{N} \tag{3}$$

Using this notation it is possible to describe the algorithm concisely, as we do in Figure 3. The scheme depends on two free parameters Π_{cross} and Π_{mut} which define the crossover and mutation rates. Let us note that the

Initialization:

Compute a random population with M members $P = \{s^{(j)} \in S^l, j=0,1,..,M-1\}$ where each string is built randomly choosing symbols of S.

Evaluation:

Evaluate the fitness $f^{(i)} = f(s^{(i)})$; compute the rescaled fitness $\bar{f}^{(j)}$:

$$f_{\min} = \min\{f^{(j)}; j=0,1,..,M-1\};\quad f_{\max} = \max\{f^{(j)}; j=0,1,..,M-1\};\quad \bar{f}^{(i)} = \frac{f^{(i)} - f_{\min}}{f_{\max} - f_{\min}}$$

and evaluate the 'running' fitness $F^{(j)}$:

$$F^{(-1)} = 0;\quad F^{(0)} = \bar{f}^{(0)};\quad F^{(j)} = F^{(j-1)} + \bar{f}^{(j)}$$

Test:

If the population P contains one ore more individuals realizing the optimization goal within the requested tolerance, stop the execution.

Selection (stochastic):

Build a new population $Q=\{q^{(j)}; j=0,1,..,M-1\}$ of individuals of P such that the probability that an individual $q \in Q$ is member of P is given by $\frac{f(q)}{\sum_{p \in P} f(p)}$:

for i = 0,1, .. ,M-1
  let w = rand(F^(M-1))
  search for a j such that F^(j-1) < w ≤ F^(j) and let q^(i) = s^(j)

Reproduction (uniform crossover):

Choose $\frac{N}{2}$ distinct pairs $(q^{(i)}, q^{(j)})$ using the N individuals of Q. For each pair build, with probability Π_{cross}, a new pair of offsprings mixing the parent's genes, otherwise copy the parents to the offsprings:

for i = 0,1, .., (M-1)/2
  if rand(1) < Π_cross
    for j = 0,1, .., l-1
      if rand(1) < 0.5
        $\bar{q}^{(2i)}{}_{(j)} = q^{(2i)}{}_{(j)};\quad \bar{q}^{(2i+1)}{}_{(j)} = q^{(2i+1)}{}_{(j)}$
      else
        $\bar{q}^{(2i)}{}_{(j)} = q^{(2i+1)}{}_{(j)};\quad \bar{q}^{(2i+1)}{}_{(j)} = q^{(2i)}{}_{(j)}$
  else
    $\bar{q}^{(2i)} = q^{(2i)};\quad \bar{q}^{(2i+1)} = q^{(2i+1)}$

Mutation:

In each new individual $\bar{q}^{(j)}$ change, with probability Π_{mut}, each gene $\bar{q}^{(j)}{}_{(i)}$ with a randomly choosen gene of S different from $\bar{q}^{(j)}{}_{(i)}$. Let us denote the new population Q'.

Replacement:

Replace the population P with the newly computed one Q'.

Iteration:

Go to the step 'Evaluation'.

Figure 3. Pseudocode for the 'classical' version of GA. In this code rand (x) returns a random number in the range [0,x). We rescale the fitness to enforce the difference among individuals and use the running fitness to implement an efficient (order log(M)) fitness-proportional selection

Table 2. GA qualitative behaviour

Mutation probability (Π_{mut})	Crossover probability (Π_{cross})		
	Null	Low	High
Null	No dynamics	Slow GA with high probability of premature convergence	Fast GA with high probability of premature convergence
Low $\left(\approx \frac{1}{l}\right)$	Random search with slow dynamics	Slowly converging GA	**Good performing GA**
High $\left(\gg \frac{1}{l}\right)$	Random search with fast dynamics	Perturbed random search	Bad performing GA

performance of the algorithm does not critically depend on their values. In Table 2 we describe the qualitative behaviour when one changes these two parameters.

The theoretical analysis of the GAs is based on the notion of schema, i.e., a template where certain loci are allowed to contain any symbol; these loci are marked with the 'don't care' or 'wild card' symbol '*'. As an example the schema '*01*11' contains '101111', '001111', '101011', '001011'. Let us denote by H the set of strings matching the template. The main result about the schema dynamics was given by Holland[11], who demonstrated that the number $m(H,t)$ of representatives of a schema H in a population at step t satisfies the following inequality:

$$m(H,t+1) \geqslant m(H,t)\frac{f(H)}{\bar{f}} \times \left[1 - \Pi_{cross}\frac{\delta(H)}{l-1} - o(H)\Pi_{mut}\right] \quad (4)$$

In the above equation, $f(H)$ is the average fitness at step t of the individuals in P that are members of H:

$$f(H) = \frac{1}{|H \cap P|}\sum_{s \in (H \cap P)} f(s) \quad (5)$$

and $\bar{f}$ is the average fitness of the population

$$\bar{f} = \frac{1}{|P|}\sum_{s \in P} f(s) \quad (6)$$

while $\delta(H)$ and $o(H)$ measure the 'disruptive' effect of the crossover and mutation operators. From Equation (4) one derives two fundamental implications about GAs:

- The dynamics of GAs is such that the proportion of schemas with fitness higher than the average fitness increases exponentially within the population (because of the factor $f(H)/\bar{f}$, see Reference 12).
- Without crossover and mutation operators, which enforce the diversity among individuals and introduce new schemas in the population, a GA algorithm would converge to suboptimal configurations, where typically all the individuals are crowded into a peak; this is analogous to a local minimum stop. This pathological behaviour, which is known as premature convergence, may be hidden in a GA if the mutation or crossover rates are insufficient[18].

We tested the 'classical GA' on a set of combinatorial problems described by the N-k model[19], which will be sketched in the following pages.

TEST PROBLEMS: QAP AND N-k MODEL

For concreteness of presentation, we briefly illustrate two optimization problems that are paradigmatic in different fields: the quadratic assignment problem (QAP) and the N-k model.

The quadratic assignment problem

QAP is NP-hard (it contains the travelling salesman problem as a special case) and it is the natural expression for practical problems ranging from the appropriate placement of modules on a chip, to the distribution of service centres in a given organization or in a region.

In QAP the configurations are given by the assignment of N units to N locations. The distance between location i and j is a_{ij} and there is a flow from unit r to unit s equal to b_{rs}. The function to be minimized is the total cost of the 'transfers', given by the sum of products distance × flow, i.e.,

$$f = \sum_{i=1}^{N}\sum_{j=1}^{N} a_{ij}b_{\phi(i)\phi(j)} \quad (7)$$

where ϕ is the assignment of units to locations, i.e., a permutation of $\{1,2,\ldots,N\}$.

The benchmark problems are generated according to the 'random problem' algorithm described in Reference 6. The elements of the (symmetric and with a null-diagonal) matrix a_{ij} and b_{ij} are produced with a specific pseudo-random number generator in the range [0,99]. The suboptimal values listed in Reference 6 ('provably or probably' the best solutions) are used for the algorithm termination.

The implementation of the TABU list is chosen in order to be convenient when the size of the list L_T is changed. An elementary movement (exchange of two units) is taboo if and only if it assigns both units to locations that they had occupied within the L_T most recent iterations.

The necessary information is recorded in an $N \times N$ matrix containing the latest occupation time for each unit and location.

The *N–k* model

From a genetic point of view, it is relevant to study the situations that arise when the different genes of an individual contribute to determine the fitness through complex mutual interactions. 'Epistasis' is the biological term that is used for describing these interactions. Pictorially, low epistasis corresponds to smooth fitness surfaces, high epistasis to wrinkled surfaces. The *N–k* model is a convenient benchmark for analysing 'tunably rugged' landscapes.

In the *N–k* model the fitness *f* of an individual *s* (represented by a binary string with *N* bits) is given by:

$$f(s) = \sum_{j=1}^{N} f_j(s_j, s_{j1}, s_{j2}, \ldots, s_{jk}) \qquad (8)$$

where the functions f_j depend on s_j and on *k* other genes that in our case are randomly chosen.

When $k = 0$ the model has no epistasis (linear model), when $k = N$ the model is fully epistatic and the optimization problem becomes extremely difficult. If we fix *N* and *k* and we choose randomly the functions f_j (filling a look-up table with random values), we hit two targets: i) we can easily build different models, by changing the seed of the random number generator, and ii) we can use the theoretical results about the properties of the fitness function[20]. In fact, because of the Law of Large Numbers[21] the fitness values tend to be Gaussianly distributed for large *N* values (see for example Figure 4 for the distribution and the fitted Gaussian in the case $N = 20$ and $k = 3$) and therefore there is a good estimate of the relative ranking of the individuals with respect to the best one.

The results obtained with the 'classical GA' for a low and medium epistatic model are shown in Figure 5, where we plot the average number of generations versus the population size. In the parallel version, where the individuals evolve simultaneously, the number of generations is proportional to the actual execution time. It should be noted that the parallel execution varies with the population size and exhibits a minimum at around 20 individuals.

PARALLELISM IN A PROBABILISTIC FRAMEWORK

This paper is dedicated to large scale parallelism in combinatorial optimization. We present some significant parallelization schemes, under the constraint that the speed-up should be close to the number of processors *P* up to large numbers of them. The two optimization algorithms considered are representative of two extreme situations. In one case (TABU search) the processors are executing independent searches. In the other case (GA) the interaction between different search processes (the various genotypes) is strong, because the generation of new candidate points depends on the consideration of many members of the population (for how many, see the three possibilities presented in the section on parallel and parallelized GAs). We are aware of different proposals for both parallel GA and TABU algorithms, differing both in the approach and in technical details related to different multiprocessors, but we concentrate here on some significant and reasonably general schemes.

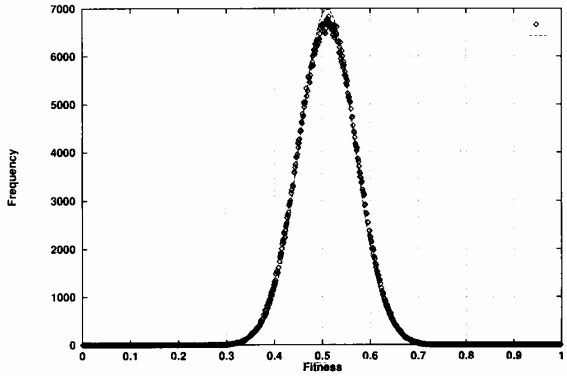

Figure 4. Distribution of fitness values and Gaussian fit for the model with N = *20 and* k = *3*

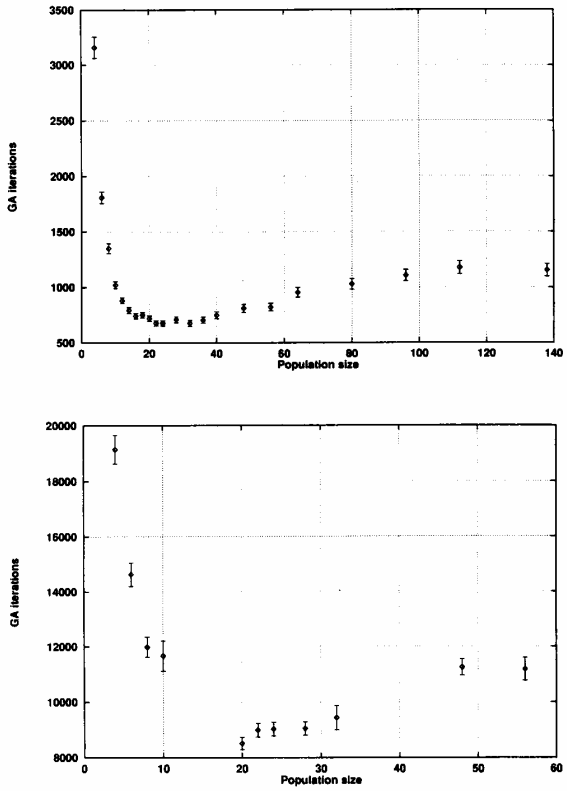

Figure 5. Number of generations needed for the 'classical' GA to find the best individual. **a,** *Low epistatic case with* N = *20 and* k = *3;* **b,** *medium epistatic case with* N = *20 and* k = *6. 1024 tests for each point. In the second case a lower number of sizes has been tested. In both cases* $\Pi_{mut} = 0.05$ *and* $\Pi_{cross} = 1.0$

In this section we introduce some tools from probability and statistics that will be used in the following analysis.

Parallelism and exponential distribution

For simplicity, let us start with a parallel scheme that considers many independent search processes. Given an

instance of a complex optimization problem (with a rough 'fitness surface' characterized by a large number of uncorrelated local minima) a solution process with different (randomly generated) initial configurations can be characterized by the probability that the optimal solution is found after iteration t. A 'time' unit corresponds to one complete move in the search, i.e., neighbourhood evaluation, choice and 'book-keeping' operations. For convenience, let us consider the cumulative probability of failing to achieve the global optimum before time t : $p_{sf}(t)$ (probability for sequential failure). In a parallel version with P processes, each running an independent version of the search, one has failure at time t if and only if all processes fail at time t (clearly the search is terminated when one process finds the desired solution), therefore the probability $p_{pf}(t,P)$ of a parallel failure with P processors is:

$$p_{pf}(t, P) = p_{sf}(t)^P \tag{9}$$

Maximum speed-up with P processors (in the probabilistic framework) is attained if the probability for a parallel failure at time t is the same as that for a sequential failure at time Pt. Maximum speed-up is the requirement that a single processor has to work for a number of steps equal to the total for all processors in order to reach the same probability of success. Remembering Equation (9), the requirement becomes:

$$p_{sf}(t)^P = p_{sf}(Pt) \tag{10}$$

But if Equation (10) is valid, then $p_{sf}(P1) = p_{sf}(1)^P$, and, after renaming variables and expressing $p_{sf}(1) = e^{-1/\tau}$ (with $\tau = -1/\log(p_{sf}(1))$) one obtains the general form for a probability distribution achieving maximum speed-up:

$$p_{sf}(t) = e^{-t/\tau} \tag{11}$$

This is the well known exponential distribution, which describes the 'waiting time' for a stochastic event, when the single trials are independent (see for example Reference 22). Both the average and the standard deviation of this distribution are equal to τ. The exponential distribution is a faithful description for the probability of failure produced by an iterated random sampling procedure, where τ is equal to the total number of points in the search space. In fact, if C is the total number of configurations, the probability of success in a single trial is $1/C$, the probability of failure is $1 - 1/C \approx e^{-1/C}$, and $e^{-t/C}$ after t trials. Remembering Equation (11), the parameter τ coincides with the number of states C. C usually is very large. For example, in the 100-unit QAP, $C = 100! \approx 9.332 \times 10^{157}$.

The TABU search is not an iterated sampling technique. The transition to a new point is a complex result of the recent history of the search process (so that the process is not even Markovian). Nonetheless, the behaviour of a TABU search on a time scale that is larger than the TABU list size is described by Equation (11) with a good agreement (if cycles are avoided !). After an initial period, during which the system 'remembers' the initial configuration, the failure probability begins to follow the exponential distribution. The initial period is related to the time for reaching the first local minimum, as will be shown in the dedicated section.

Autoregressive processes: AR(1)

Although not directly applicable to describe the evolution of a TABU search, the theory of Markov processes is an important statistical tool for describing the structure of a fitness surface. As an example, the description is useful for choosing appropriate mutation and recombination operators in GA, or to estimate the numbers of local minima in a problem, the average number of steps from a random configuration to the nearest local minimum, etc.

In the absence of *a priori* information, some indication about the structure of an unknown fitness surface can be derived by generating points on the surface, subjecting them to a random walk, and observing the time series of fitness values[20].

In many cases (for example for the N-k model and for our random QAP) the time series can be suitably described by a first-order autoregressive process AR(1), sometimes called the Markov process. If $Z(t)$ is a purely random variable with mean zero and variance σ_Z^2, then $f(t)$ is an autoregressive process of order one if:

$$f(t) = \alpha f(t-1) + Z(t) \tag{12}$$

If the mean μ_f of the function is different from 0, one must substitute $f(t)$ with its displacement with respect to the mean, i.e., $\tilde{f}(t) = f(t) - \mu$. From the theory[21] one derives the following relation between the variance of $f(t)$, that of $Z(t)$ and the parameter α:

$$Var[f(t)] \equiv \alpha_f^2 = \sigma_Z^2/(1 - \alpha^2) \tag{13}$$

The conditional distribution for the function values of the neighbours of a point s, given that its function value is $f(s)$, is normal (Gaussian) with the following mean and variance:

$$E[f(s'), s' \in \Omega(s) \mid f(s)] = \mu_f + \alpha(f(s) - \mu_f) \tag{14}$$

$$Var[f(s'), s' \in \Omega(s) \mid f(s)] = \sigma_Z^2 \tag{15}$$

Note that the conditional mean is between the actual value $f(s)$ and the mean of $f(s)$ over the surface μ_f, with α regulating its position. In addition, the autocovariance and autocorrelation functions of the time series behave like decaying exponentials and are given by:

$$\text{acv.}f(h) \equiv E[f(t)f(t+h)] = \alpha^h \sigma_f^2 \tag{16}$$

$$\begin{aligned}\text{ac.}f(h) &\equiv E[f(t)f(t+h)]/(E[f(t)]E[f(t+h)]) \\ &= \alpha^h = e^{-h/\tau_a}\end{aligned} \tag{17}$$

where $\tau_a = -1/\log\alpha$, and α is recognized as the autocorrelation between subsequent function values.

Counting local minima in QAP

The probability that a random configuration s with value $f(s)$ is a (strict) local minimum is the probability that all neighbours in $\Omega(s)$ have a higher function value.

Let us test the AR(1) model on the QAP problem (for the N-k model, see Reference 20). In a good approximation, the distribution of function values for the 'random' QAP problems that we consider is Gaussian (this is confirmed by the Kolmogorov-Smirnov test[23]), and its parameters can easily be estimated by randomly sampling function values. The parameters for the conditional distribution of neighbouring values are derived from

Equations (14) and (15), after calculating σ_Z from σ_f and α using Equation (13).

The probability that a single neighbour is higher is the integral of the conditional distribution from $f(s)$ to infinity. This has to be raised to a power given by the number of neighbours to obtain the conditional probability that all of them are higher, i.e., that s is a local minimum. The cardinality of $\Omega(s)$ is $N(N-1)/2$, equal to the number of possible exchanges.

Finally, the probability of local minima is obtained by integrating the probability density function of fitness values times the conditional probability of a local minimum, given a fitness value. As expected, the local minima are distributed mainly on the extreme tail of the distribution, close to the absolute minimum. In Figure 6 we show the distribution of function values for the $N = 100$ instance of QAP, the probability that a value is a local minimum, and the probability density function (PDF) for the local minima (the PDF for function values multiplied by the probability that the value is a local minimum).

From the number of local minima we can estimate the average numer of steps from an initial starting point to the 'nearest' local minimum. Let us define as an attraction basin the set of points that will end up in a local minimum using the steepest descent procedure. Assuming an equal size for the different attraction basins, the number of points in a basin will be given by $N!/n_{\text{locmin}}$, in which n_{locmin} is the total number of local minima. Because with a number of iterations l the number of reachable points is approximately $(N(N-1)/2)^l$, we can derive a crude approximation to the maximum number of steps to the nearest local minimum as $l_{\max} \approx \log(basin)/\log(N(N-1)/2)$. The approximation is rough, both because the size of the basin tends to be larger for local minima with lower function values and because the estimate for the number of reachable points fails for large l (the same points can be reached with different sequences of exchanges).

Various theoretical estimates have been calculated for different sizes of the QAP problem and, when possible with the available CPU time, compared with experimental results, see Table 3. A comparison with the experimental fraction of local minima has shown remarkably good agreement with the calculation, providing a consistency check for the model. The experimental comparison is problematic for large N values, for example, for $N = 100$ the estimated probability for a local minimum is $\approx 3.65 \times 10^{-84}$.

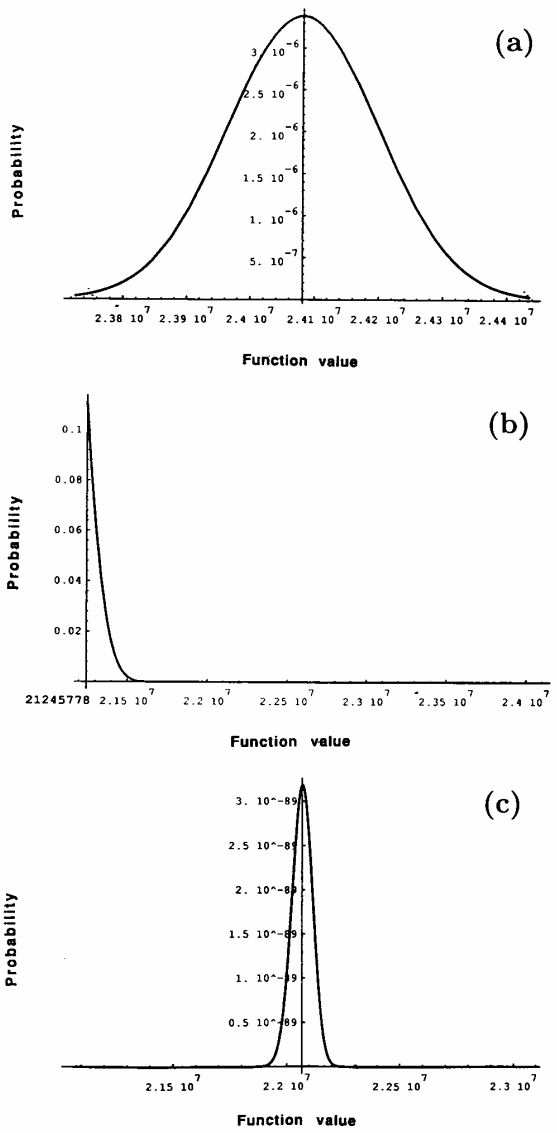

Figure 6. QAP with 100 units: **a,** *probability density function for the distribution of function values;* **b,** *probability that a value is a local minimum;* **c,** *PDF for the distribution of local minima. Local minima are concentrated near the extreme tail of the distribution (the value 21 245 778 is the value used for terminating the search)*

PARALLEL SCHEMES AND CONCRETE EXAMPLES

TABU with hashing: independent search

The only requirement that we make is that a single processor is capable of supporting a complete search process of the problem. If this is not the case, the parallel machine will need to be subdivided into interconnected clusters of processors.

In Figure 7 we plot the success probability of TABU with hashing as a function of time (for the case $N = 10$) and the fit with the exponential distribution. Cases with higher N values are qualitatively similar: the first local minimum is found after a number of iterations that is very small with respect to the total required for finding the global optimum, so that the exponential distribution provides a suitable fit apart from the very first iterations. The hypothesis for the maximum efficiency of Equation (10) is satisfied, allowing an efficient use of parallelism. The probability density function $p_{1st}(t)$ for the probability of finding the first local minimum at time t is plotted in Figure 8, for N ranging from 10 to 30.

The PDFs for the success probability of the parallel implementation of TABU with hashing is shown in Figure 9 (for $N = 10$ and a number of processors P ranging from 8 to 64). Comparing the actual results with the theoretical distributions derived in the hypothesis of a

Table 3. Statistical estimates (*t*) and comparison with experimental data (*e*) for various sizes *N* of the QAP problem

N	Probability of local minimum (*t*)	Probability of local minimum (e)	Size of basin (*t*)	Steps to nearest local minimum (*t*)	Steps to nearest local minimum (e)	Number of local minima (*t*)
10	1.25×10^{-4}	1.14×10^{-4} ($5 \times 10^{+5}$ trials)	$8.0 \times 10^{+3}$	2.36	5.35 (sdev = 2.0)	454
12	9.38×10^{-6}	8×10^{-6} (10^{+6} trials)	$1.06 \times 10^{+5}$	2.76	6.73 (sdev = 1.8)	4493
15	2.22×10^{-7}	N.A.	$4.49 \times 10^{+6}$	3.29	8.0 (sdev = 2.1)	290826
20	3.94×10^{-10}	N.A.	$2.53 \times 10^{+9}$	4.12	11.3 (sdev = 2.4)	$9.60 \times 10^{+8}$
30	1.57×10^{-17}	N.A.	$6.33 \times 10^{+16}$	6.36	16.87 (sdev = 3.3)	$4.18 \times 10^{+15}$
100	3.65×10^{-84}	N.A. Difficult to estimate!	$2.73 \times 10^{+83}$	22.58	59.00 (sdev = 6.8)	$3.41 \times 10^{+74}$

N.A. Not available due to excessive computing times required

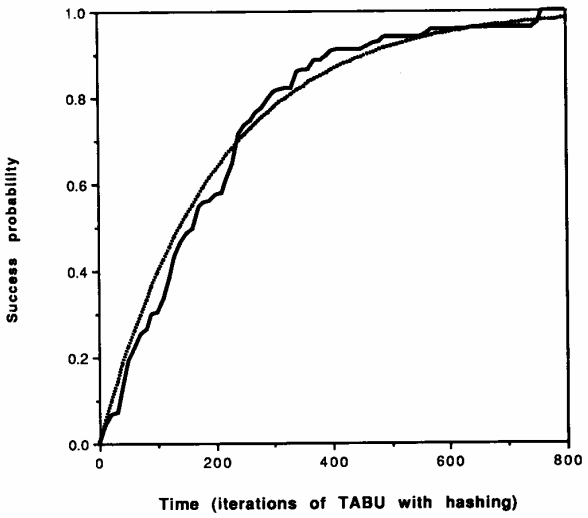

Figure 7. Success probability of TABU with hashing as a function of time, for the N = 10 QAP problem. Fraction of successes on 1000 independent runs with different initial points (continuous line) and best fit with the exponential distribution $p(t) = 1 - exp(-t/\tau)$ *(dotted line).* $\tau = 200.674$

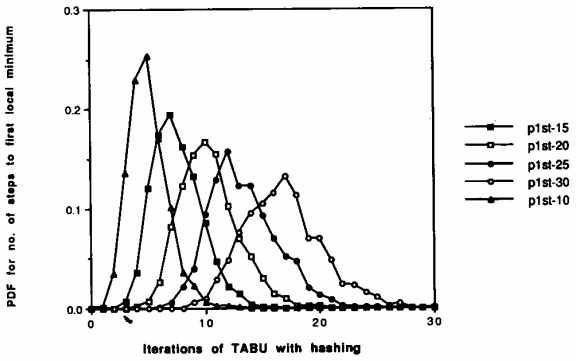

Figure 8. Estimated PDF for the number of steepest descent iterations to reach the local minimum (QAP). Different curves are for different problem sizes (from N = 10 to 30). Statistics on 1000 random starts for each problem size

'pure' exponential distribution, one can observe an increasing difference between the two distributions for large numbers of processors, when the average solution times become comparable with the time for reaching the first local minimum. Similar conclusions are derived by observing the behaviour of the average solution time as a function of the number of processors in Table 4. As a refinement to the exponential distribution one can derive the cumulative probability $p_s(t)$ of success at time t with the following integral:

$$p_s(t) = \int_0^t dt' \; p_{1st}(t') \, p_s(t \mid 1st \text{ at } t') \qquad (18)$$

where the conditional probability $p_s(t \mid 1st \text{ at } t')$ of success at time t, given that the first local minimum is found at t', is modelled by an exponential distribution ($p_s(t \mid 1st \text{ at } t') \approx 1 - e^{-t/\tau}$) and $p_{1st}(t')$ is estimated experimentally.

One important point about the parallel implementation of TABU is that one does not need complex parallelization schemes because that based on independent searches already reaches an efficiency that is very close to one, even for large numbers of processors, with a limited effort with respect to the sequential version. A similar result has been found in Reference 24 for simulated annealing, and in Reference 6, for the TABU variation considered.

Parallel GA or parallelized GA?

The parallelization scheme based on independent searches described for TABU can be applied to GA also, provided that each processor is capable of supporting an entire population (e.g. a large 'grain size' MIMD machine). In fact, the approximately exponential behaviour of the failure probability is evident from Figure 10.

But the 'fine scale' parallelism of GA due to its use of a population of interacting individuals admits 'fine grain'

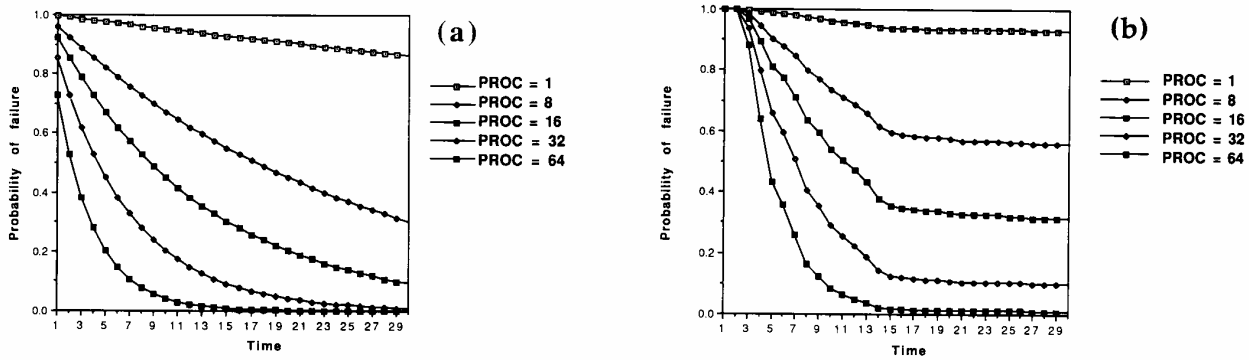

Figure 9. Probability of failure for the parallel TABU with hashing search: **a,** *'Pure' exponential distribution;* **b,** *experimental data*

Table 4. Average success time as a function of the number of processors *P*, for the *N* = 10 QAP. In parentheses are the expected times in the hypothesis of maximum speed-up

	P = 1	*P* = 8	*P* = 16	*P* = 32	*P* = 64	*P* = 128	*P* = 256	*P* = 512
N = 10	200.67	34.15 (25.08)	18.54 (12.54)	9.70 (6.27)	5.33 (3.13)	3.62 (1.56)	2.82 (0.87)	2.38 (0.39)

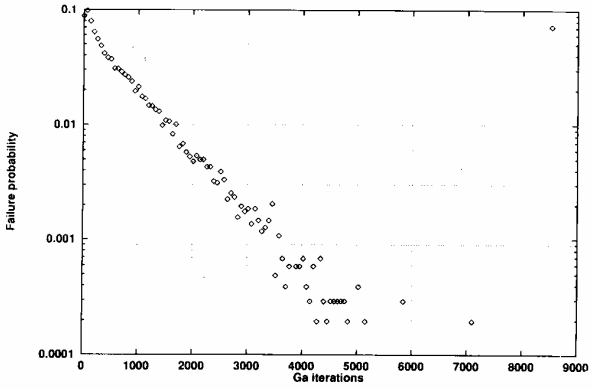

Figure 10. Probability of failure for the 'classical' GA (logarithmic scale)

implementations (one individual in each processor) with a higher degree of interaction between the different processes. In a way, GAs mimic the fully parallel mechanisms of the biological world so that the parallel implementation is even more 'natural' than the sequential simulation.

At this point, it is important to distinguish between two different possibilities: parallel GAs, i.e. 'intrinsically parallel' algorithms suggested by the architecture of specific parallel machines (like interconnected meshes of processors), and parallelized GAs, i.e., the parallel implementation of algorithms like the 'classical GA'. If one adopts the engineering (versus biology) point of view of minimizing solution times, the choice between the various possibilities is not trivial. For example, the advantage of communicating with nearest neighbours in a grid architecture may well be overbalanced by the slower propagation of information during the evolution. The two options will be illustrated in the following sections.

Parallelized GA

The 'classical GA' admits a straightforward parallelization. In fact, each stage of the algorithm operates almost independently on the single individuals (evaluation, mutation and replacement operate independently, while the crossover depends on two individuals), except for the selection phase, which is effected by a 'global interaction' of the whole population. The communication overhead caused by this global communication may seriously reduce the parallel efficiency of the implementation (see for example References 25 and 26), unless a high speed global communication mechanism is available in the parallel machine used. On MIMD architectures and SIMD architectures with sparse (e.g. grid) interconnections allowing only nearest neighbours communications (with low-speed channels) this condition is not satisfied so that scaling the implementation to very large numbers of processors may be problematic. On the other hand, the scheme is sufficiently efficient in a machine like the Connection Machine, with dedicated hardware for general communication patterns (we used a CM2 with 4096 PEs). In the selection phase each individual (handled by a processor of the CM) extracts a random number and uses an order log(*N*) search algorithm to find the first individual with running fitness lower than the extracted number (using get-like operations), as is realized with the cstar code in Figure 11. We use the CM also to compute the fitness in the *N–k* model. Given the limited amount of memory on each processor (256 kbit), the look-up table describing the fitness function is larger than the memory associated with the single PE for large problem sizes. From Equation (8) every f_j depends on $k + 1$ bits, giving $2^{(k+1)}$ distinct values. Because every value is represented with 32 bits and because we have N different f_js the total memory requirement is $32 \times N \times 2^{(k+1)}$ bits. As soon as this space gets larger than the available 256 kbit, it is necessary to distribute the table among the PEs and to use

```
/* Selection phase for 'classical GA': each individual computes the running fitness, then extracts   */
/*                                     a random number and searches, with a bisection algorithm,     */
/*                                     for the first individual with running fitness lower than the  */
/*                                     extracted number. This procedure is executed two times giving */
/*                                     the pair of mating individuals, whose strings will be stored  */
/*                                     in str1 e str2 for the further stages of crossover and mutation.*/

        with (Pop)
                {
                 real:Pop       F,w;
                 int:Pop        k,h;
                 bool:Pop       str1[N],str2[N];        /* the two mating individuals                 */
                 bool:Pop       s;

                 CMC_sendaddr_t:Pop     addr;

                 F  = 0;                                 /* initialize                                 */
                 s  = 0;
                 boolset (str1, s, N);
                 boolset (str2, s, N);

                 where (active)
                       {
                        F = scan (fitness,               /* compute the running fitness                */
                                  0,
                                  CMC_combiner_add,
                                  CMC_upward,
                                  CMC_none,
                                  CMC_no_field,
                                  CMC_exclusive);

                        maxF = [POPSIZE-1]F + [POPSIZE-1]fitness;
                        F   /= maxF;                                    /* rescale in the range [0,1]  */

                        i = POPSIZE >> 1;
                        k = 0;
                        frand (&w);             /* extract a random number in the range  [0,1]        */
                        while (i)               /* search for a individual with running fitness       */
                              {                 /* just below the extracted number                    */
                               h = k | i;
                               where (w > [h]F)         /* this is a 'get' operation                  */
                                     k = h;
                               i >>= 1;
                              }

                        addr = make_send_address(Pop, k);       /* compute the address and            */
                        CM_get_1L       (str1,                  /* 'get' the first value parent       */
                                         &addr,
                                         str,
                                         N);

                        i = POPSIZE >> 1;       /* do the same for the second parent                  */
                        k = 0;
                        frand (&w);
                        while (i)
                              {
                               h = k | i;
                               where (w > [h]F)
                                     k = h;
                               i >>= 1;
                              }

                        addr = make_send_address(Pop, k);
                        CM_get_1L       (str2,                  /* 'get' the second parent            */
                                         &addr,
                                         str,
                                         N);
                        boolcpy(str, str2, N);
                       }
                }
```

Figure 11. cstar code for the selection phase of the parallelized 'classical' GA. The notation [var1]var2 is a left indexing and represents the access of the variable var2 stored in the PE var1. For further detail see Reference 27

get-operations to access it. Figure 12 contains the cstar code for the fitness function. The performance of this implementation is shown in Table 5, which reports the timing for the different sections of the critical code.

Parallel GA

The amount of communications needed during the selection phase of the 'classical GA' causes a bottleneck in the parallel implementation, which may seriously degrade the efficiency, especially if the complexity of the fitness function evaluation is limited.

In this event, it is appropriate to modify the genetic paradigm to make it more suited to a parallel architecture and this can be done by simulating nature in a more strict manner[28]. The parallel GA partitions the population into groups of individuals (overlapping or non-overlapping), where the genetic evolution takes place (sometimes

```
#define KNUM (1 << (K+1))

shape                   [POPSIZE]Pop;          /* population geometry (one-dimensional)        */
bool:Pop                str[N];                /* N bit string representing individual         */
float:Pop               fitness;               /* individual fitness                           */

shape                   [N][KNUM]FitTable;     /* fitness table geometry (for N*2^(K+1) values)*/
float:FitTable          val;                   /* fitness values                               */

int                     links[N][K+1];         /* links[j] contains the links of the j-th bit  */

InitializeFitnessTables ()                     /* build randomly a N-K model                   */

{

int                     a[N];
int                     i,j,k,w;

        for (i = 0; i < N; ++i)
                {
                  for (j = 0; j < N; ++j)
                        a[j] = 0;
                  for (j = 0; j < (K+1); ++j)
                        links[i][j] = 0;

                  a[i] = 1;                    /* each gene is linked to itself                */
                  k = 0;
                  while (k < K)                /* compute the links of the i-th gene           */
                        {
                          w = rand() % N;      /* rand returns a scalar integer value          */
                          if (a[w])            /* if the link is already existing              */
                                continue;
                          a[w] = 1;
                          ++k;
                        }

                  k = 0;
                  for (j = 0; j < N; ++j)
                        if (a[j])
                                links[i][k++] = j;
                }

        with (FitTable)
                val = frand();                 /* frand returns a random parallel real value   */
                                               /* in the range [0,1)                           */
}

EvalFitness ()          /* parallel computation of the individual fitness for the N-K model    */

{

int             i,j;

int:Pop         n;
float:Pop       w;

union                                          /* k is used to compute the row index in the    */
        {                                      /* fitness table (the column index is given by  */
          bool          b[K+1];                /* the locus of the gene). The i-th bit of k    */
          unsigned      int     i;             /* is the value of the i-th linked gene         */
        }
                :Pop    k;
CMC_sendaddr_t  :Pop    addr;


        fitness = 0;
        for (i = 0; i < N; ++i)
                {
                  k.i = 0;
                  for (j = 0; j < (K+1); ++j)          /* compute the row index in the fitness table   */
                        k.b[j] = str[links[i][j]];
                  n = i;
                  addr = make_send_address(FitTable, n, k.i);   /* compute the address of the PE storing*/
                                                                /* the [n,k.i] fitness value and        */
                  w = get(addr, &val, CMC_collisions);          /* get it                               */
                  fitness += w;
                }
        fitness /= (float) N;                          /* normalize the fitness                */

}
```

Figure 12. cstar code for the parallel fitness computation in the N–k model

Table 5. Timing of the two critical sections in the parallelized 'classical GA'. These values were obtained by running the algorithm on a CM2 with 4096 processors and with a population of 4096 individuals, $\Pi_{mut} = 0.05$ and $\Pi_{cross} = 1.0$. The fitness in the N-k model is evaluated by distributing the look-up table among $N \cdot 2^{(k+1)}$ virtual PEs. 'N.A.' indicates a failure in the allocation of the requested number of virtual processors (due to the high number) because of the excess memory requirements. When the number of processors needed is larger than the existing physical processors, the software can simulate 'virtual processors' with time and memory slicing

CM2 timing							
N-k model		CM busy time (ms)		N-k model		CM busy time (ms)	
N	k	Eval fitness	Evolve	N	k	Eval fitness	Evolve
64	2	140	103	128	2	279	152
64	4	172	103	128	4	345	152
64	6	219	103	128	6	449	152
64	8	286	103	128	8	607	152
64	10	413	103	128	10	922	152
64	12	567	103	128	12	1449	152
64	14	939	103	128	14	3342	152
64	16	2785	103	128	16	N.A.	N.A.
32	2	70	79	256	2	559	250
32	4	86	79	256	4	701	250
32	6	88	79	256	6	929	250
32	8	135	79	256	8	1823	250
32	10	192	79	256	10	2180	250
32	12	187	79	256	12	4193	250
32	14	308	79	256	14	11 602	250
32	16	7781	79	256	16	N.A.	N.A.

'helped' by an explicit hill-climbing mechanism). After a given number of new generations, the groups of individuals interact and exchange the best members in each group. This scheme is well suited for MIMD architectures (where each PE evolves a group of individuals) because of the low interprocessor communication overhead. An algorithm of this type has been proposed by Mühlenbein *et al.* and has been applied to a wide range of problems (see for example References 29–31). An advantage of Mühlenbein's approach is the intrinsic asynchronous evolution of the 'islands' of individuals: each group evolves without any external synchronization. This is in fact an example of the 'loosely synchronous problems' described in Reference 25 that are optimally suited for implementations on MIMD machines. In addition, when the fitness computation is a very complex task or does not admit an efficient parallelization, this is the only feasible method to implement a GA on a parallel architecture. A sketch of the 'building blocks' of the algorithm is given in Figure 13.

A parallel GA that implements a similar scheme using only interactions between nearest neighbours is proposed in Reference 32 for a SIMD machine. The scheme employs fully overlapping groups of individuals, as summarized in Figure 14.

Both variants may suffer from the point of view of resource optimization: because the population is spatially

Initialization:
Create an initial population by generating a set of groups of individuals.

Local Evolution:
local hill-climbing:
each individual does local hill-climbing during a fixed time period.
local genetic-like evolution:
every group executes a 'classical GA'.

Global Evolution:
After a fixed number of generations each group sends the best individuals to the groups in its neighbourhood.

Termination:
if the termination condition is not satisfied, go to the 'local evolution' step

Figure 13. Skeleton of the asynchronous parallel GA

Initialization:

Create an initial population of randomly chosen individuals.

Evaluation:

Compute the fitness of each individual.

Test:

If the optimization goal is reached, stop the execution.

Selection:

Let $Q = \{q^{(j)}; j=0,1,..M-1\}$ denote the reproductive set, $P = \{s^{(j)}; j=0,1,..M-1\}$ the population and $\Omega(s)$ the neighbourhood of $s \in P$. Then an individual $q^{(j)} \in Q$ is member of $\Omega(s^{(j)})$ and it is chosen with probability $\frac{f(q^{(j)})}{\sum_{p \in \Omega(q^{(j)})} f(p)}$.

In other words, the "selection set" of a individual is given by its neighborhood and not by the whole population (as in the 'classical GA').

Reproduction:

Randomly choose an individual $s \in P$ in the neighborhood $\Omega(s^{(j)})$ and, with probability Π_{cross}, mate it with the respective individual $q^{(j)}$ in the reproductive set Q.

Mutation:

Mutate each gene with probability Π_{mut}.

Iteration

Go to step 'evaluation'.

Figure 14. Skeleton of the synchronous parallel GA

structured (e.g. on a two-dimensional grid) and because the interaction is only among nearest neighbours, the propagation of the genetic information is a diffusive process. The slow time scale of the diffusion process limits the premature convergence of the population but may do so at the price of slowing the entire search process. As an example, in the SIMD implementation of Reference 32, on a two-dimensional mesh of $N \times N$ PEs the cost to spread the genetic information over the whole population is of the order N generations. On the other hand, for the parallelized version of the 'classical GA', the cost for the same operation is one generation, at the price of a higher communication overhead ($\log(N)$ times the cost for crossing one dimension of the hypercube).

CONCLUSIONS

In this paper we presented some significant and relatively general techniques for parallel combinatorial optimization. In particular the TABU scheme, based on the use of hashing for cycle detection, admits concurrent execution of independent search streams because of the statistical distribution of the success times. Although we are aware of different parallel versions, the one presented requires a negligible effort with respect to the sequential version and reaches very high efficiency for significant optimization tasks.

GAs admit a similar parallel scheme but we can expect higher speed-up improvements, exploiting their intrinsic parallel behaviour. This can be easily done on MIMD machines with simple communication schemes as on transputer-based systems where high efficiency levels are expected, especially if most of the computation time is spent on computing the fitness function.

In fact, in addition to the solution of regularly structured problems (such as fluid dynamics in the continuum case or, in the digital case, cellular automata) it is not difficult to foresee optimization as one of the major future applications of massively parallel machines. This will be a result both of the effectiveness of the parallel implementation and the need for global optimization in many different areas. Last, but not least, let us remember the growing field of automated learning (see for example neural networks, but also more traditional AI techniques) where the optimization part is as crucial both during learning and in the definition of architectures suitable for a proper generalization. Parallel implementation (with teraflop or more powerful machines) is in many cases the only way to reach a level of performance comparable to that of humans (in tasks like voice interaction, navigation, vision, etc.).

ACKNOWLEDGEMENTS

The authors would like to thank S. Struthers for her kind assistance and the I.N.F.N. group of the University of Parma for allowing the use of their hardware facilities.

REFERENCES

1 **Garey, M R and Johnson, D S** *Computers and Intractability: A Guide to the Theory of NP-Completeness* Freeman (1979)

2 **Rawlins, G J E (Ed)** *Foundations of Genetic Algorithms* Morgan Kaufmann, San Mateo, CA, USA (1991)
3 **Glover, F, McMillan, C and Novick, B** 'Interactive decision software and computer graphics for architectural and space planning' *Ann. Oper. Res.* Vol 5 (1985) pp 557–573
4 **Glover, F** 'Tabu search – part I' *ORSA J. Comput.* Vol 1 No 3 (1989) pp 190–206
5 **Glover, F** 'Tabu search – part II' *ORSA J. Comput.* Vol 2 No 1 (1990) pp 4–32
6 **Taillard, E** 'Robust taboo search for the quadratic assignment problem' *Parallel Comput.* Vol 17 (1991) pp 443–455
7 **Aho, A V, Hopcroft, J E and Ullman, J D** *Data Structures and Algorithms* Addison-Wesley, Reading, MA, USA (1985)
8 **Kirkpatrick, S, Gelatt, C D and Vecchi, M P** 'Optimization by simulated annealing' *Science* Vol 220 (1983) pp 671
9 **Moscato, P and Fontanari, J F** 'Stochastic versus deterministic update in simulated annealing' *Phys. Lett. A* Vol 146 No 4 (May 1990) pp 204–208
10 **Martin, O, Otto, S W and Felten, E W** 'Large-steps Markov chains for the travelling salesman problem' *Complex Syst.* Vol 5 (1991) pp 299–326
11 **Holland, J H** *Adaptation in Natural and Artificial Systems, an Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence* University of Michigan Press, Ann Arbor (1975)
12 **Goldberg, D E** *Genetic Algorithms* Addison-Wesley, Reading, MA, USA (1989)
13 **Spiessens, P** 'Genetic algorithms – introduction, applications and extensions' *AI Memo* No. 88-19, Vrije Universiteit Brussel (1988)
14 **De Jong, K A** 'An analysis of the behavior of a class of genetic adaptive systems' *Dissert. Abstr. Intern.* Vol 36 No 10 (1975) 5140B
15 **Goldberg, D E and Deb, K** 'A comparative analysis of selection schemes used in genetic algorithms' in *Foundations of Genetic Algorithms* Morgan Kaufmann, San Mateo, CA, USA (1991)
16 **Spears, W M and De Jong, K A** 'An analysis of Multi-Point Crossover' in *Foundations of Genetic Algorithms* Morgan Kaufmann, San Mateo, CA, USA (1991)
17 **Fogel, D B and Atmar, J W** 'Comparing genetic operators with Gaussian mutations in simulated evolutionary processes using linear systems' *Biol. Cybern.* Vol 63 (1990) p 111
18 **Schaffer, J D, Eshelman, L J and Offutt, D** 'Spurious correlations and premature convergence in genetic algorithms' in *Foundations of Genetic Algorithms* Morgan Kaufmann, San Mateo, CA, USA (1991)
19 **Kaufmann, S, Weinberger, E and Perelson, A** 'Maturation of the immune response via adaptive walks on affinity landscapes' in *Theoretical Immunology* Addison-Wesley, Reading, MA, USA (1988)
20 **Weinberger, E** 'Correlated and uncorrelated fitness landscapes and how to tell the difference' *Biol. Cybern.* Vol 63 (1990) pp 325–336
21 **Papoulis, A** *Probability, Random Variables, and Stochastic Processes* McGraw-Hill, New York, USA (1965)
22 **Rozanov, Y** *Processus Aléatoires,* Éditions de Moscou, Traduction française Editions Mir, Moscow (1975)
23 **Press, W H, Flannery, B P, Teukolsky, S A and Vetterling, W T** *Numerical Recipes in C,* Cambridge University Press, UK (1988)
24 **Dodd, N** 'Slow annealing versus multiple fast annealing runs – an empirical investigation' *Parallel Comput.* Vol 16 (1990) pp 269–272
25 **Fox, G, Johnson, M, Lyzenga, G, Otto, S, Salmon, J and Walker, D.** *Solving Problems on Concurrent Processors (Vol 1)* Prentice Hall, Englewood Cliffs, NJ, USA (1988)
26 **Almasi, G F and Gottlieb, A** *Highly Parallel Computing* Benjamin/Cummings (1989)
27 *Programming in cstar* and *Programming in C/Paris* Thinking Machines Corp, Cambridge, MA (1990)
28 **Mühlenbein, H** 'Darwin's continent cycle theory and its simulation by the prisoner's dilemma' *Complex Syst.* Vol 5 (1991) pp 459–478
29 **Mühlenbein, H** 'Evolution in time and space – the parallel genetic algorithm' in *Foundations of Genetic Algorithms* Morgan Kaufmann, San Mateo, CA, USA (1991)
30 **Mühlenbein, H, Gorges-Schleuter, M and Krämer, O** 'Evolution algorithms in combinatorial optimization' *Parallel Comput.* Vol 7 (1988) p 65
31 **Mühlenbein, H, Schomisch, M and Born, J** 'The parallel genetic algorithm as function optimizer' *Parallel Comput.* Vol 17 (1991) p 619
32 **Spiessens, P and Manderick, B** 'A massively parallel genetic algorithm – Implementation and first analysis' Proc. Fourth International Conf. on Genetics Algorithms, University of California, San Diego (1991)

Roberto Battitii received BS and MS degrees in physics from Trento University, Italy (1985) and a PhD degree in the new Department of Computation and Neural Systems from Caltech, Pasadena, California in 1990 for his research in optimization methods for sub-symbolic learning systems. He then became a consultant for the industrial application of neural networks and concurrent processing and, since June 1991, has been a faculty member at the University of Trento and an associate member of INFN. His main research interest remains the mathematical analysis of massive computational tasks and their implementation with parallel architectures, especially in the areas of computer vision, optimization and neural networks.

Giampietro Tecchiolli was born in Trento, Italy in 1961. He received his degree 'cum laude' in physics from the University of Trento in 1986. Since 1987 he has been associated with INFN working in the area of computational theoretical physics. He joined IRST in 1988, where he works in the area of tools and algorithms for artificial intelligence. His research interests include discrete, continuous and functional optimization, parallel computation and algorithm complexity