Distributed Genetic Algorithms

by

Jung Y. Suh and Dirk Van Gucht

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 225

Distributed Genetic Algorithms

by

Jung Y. Suh and Dirk Van Gucht

July, 1987

To appear in *Butterfly Parallel Processing*, P. Waterman, Ed.

Distributed Genetic Algorithms

Jung Y. Suh
Dirk Van Gucht

Computer Science Department
Indiana University
Bloomington, Indiana 47405
(812) 335-6429
CSNET: jysuh@indiana, vgucht@indiana

1. Introduction

1.1. Genetic Algorithms

Suppose we have an *object space* X and a function $f : X \rightarrow R^+$ (R^+ denotes the positive real numbers) and our task is to find a global optimum for that function. *Genetic algorithms* are a class of adaptive algorithms invented by John Holland [16] to solve (or partially solve) such problems. Genetic algorithms differ from more standard search algorithms (e.g., gradient descent, controlled random search, hill-climbing, simulated annealing [3, 4, 18] etc.) in that the search is conducted using the information of a *population* of *structures* of the object space X instead of that of a single structure. The motivation for this approach is that by considering many structures as potential candidate solutions, the risk of getting trapped in a local optimum is greatly reduced. In Figure 1 we show the layout of a genetic algorithm, which we will from now on call a *standard genetic algorithm.*

```
P(t) denotes the population at time t.

t ← 0;
initialize P(t);
evaluate P(t);
while (termination condition is not satisfied)
{
    t ← t+1;
    select P(t);
    recombine P(t);
    evaluate P(t);
}
```

Figure 1. Layout of a Standard Genetic Algorithm

The initial population P(0) consists of structures of X, usually chosen at random. Alternatively, P(0) may contain heuristically chosen structures. In either case, the initial population should contain a wide variety of structures. Each structure x in P(0) is then evaluated by applying to it the function f. The genetic algorithm then enters a loop. Each iteration of that loop is called a *generation.* The new population P(t+1) is constructed in two steps, the selection and recombination steps. In the selection step, a temporary population (say $P'(t+1)$) is constructed by choosing structures in P(t) according to their relative performance. For example, if we are maximizing f, the structures with greater than average performance will be selected with higher probability than the structures with below average performance. This resembles the *survival of the*

fittest principle of natural evolution. After the selection step, the temporary population $P'(t+1)$ is *recombined.* (The resulting population is the new population $P(t+1)$.) Typically, recombination is accomplished by applying several *recombination operators*, such as *crossover*, *mutation*, *inversion* [7, 16], or *local improvement* [25], to the structures in $P'(t+1)$. After the recombination step is completed, the new population is reevaluated and a termination condition is checked for validity.

Genetic algorithms have been applied with great success by De Jong [7] to a wide variety of function optimization problems defined over object spaces of the form R^n, i.e., each structure x consists of n real numbers $x[1] \ldots x[n]$. They have also been applied to other problems such as optimization of simulations [12], image processing tasks [10], evolving production system programs for AI [24], combinatorial optimization problems [5, 6, 11, 14, 15, 23, 25] etc. This wide variety of problem domains suggests that genetic algorithms are robust and flexible optimization algorithms. They suffer a serious drawback however: their implementation as sequential algorithms on sequential machines typically run slowly when compared to problem specific optimization algorithms. On the other hand, it is clear, by looking at Figure 1, that genetic algorithms can easily be parallelized and run on multi-processor machines which would greatly improve their efficiency. It is our intent to show a variety of parallel versions of genetic algorithms, which, by the way, better resemble natural evolution, and to show the results and measured speed-up of running these algorithms as simulations on sequential machines and as real parallel algorithms on a multi-processor machine.

1.2. Parallelizing Genetic Algorithms

There have been many proposals to improve the quality and performance of standard genetic algorithms. Many of these are intended to improve the robustness of the algorithm, mainly by preventing the *premature convergence problem* [7, 8, 16] by maintaining enough diversity in the population, either by normalizing the performance value of the structures in the population [1] or by introducing random noise systematically [21]. Another improvement of genetic algorithms resulted from the proposals indicating that domain specific knowledge could easily be incorporated in the recombination operators of genetic algorithms [14, 15, 25]. Still another proposal indicating that it is sometimes sufficient to provide approximate, but typically quickly obtained, function evaluations to the algorithm resulted in dramatic speed-ups of the algorithm in problem domains such as image processing [10]. In this paper, we readdress the problem of speeding up genetic algorithms by parallelizing them†. As one can observe, by looking at Figure 1, one can easily devise a parallel version of a standard genetic algorithm by considering a pool of processors which perform function evaluations and recombination operations and another processor which is responsible for assigning structures to the processors for evaluation and recombination and which furthermore performs the selection step of the genetic algorithm (in fact this is close to one of the algorithms introduced by Grefenstette [13]). We will show that one can go a lot further by also parallelizing the recombination step and the selection step of the algorithm. In Section 2, we propose a parallel version of a standard genetic algorithm in which the evaluation step and the recombination step are parallelized. We will call this algorithm the *centralized genetic algorithm* since it still uses central control because the selection step is performed by a master processor which also synchronizes the actions of the processors which perform the evaluations and recombination operations. In Section 3, we propose a framework in which genetic algorithms become totally distributed algorithms which we will call *distributed genetic algorithms.* This is accomplished by replacing the selection step by local selection routines which are distributed over the processors which already contain routines for evaluation and recombination. We will furthermore argue in Section 3 that distributed genetic algorithms yield similar performance as standard genetic algorithms, that their implementation is straightforward, uniform and natural, that they are reliable algorithms, that they allow for effects

† This problem has been considered by other researchers such as Grefenstette [13].

not possible in synchronized implementations, and that they offer more tuning opportunities to control problems, such as the premature convergence problem, than standard genetic algorithms. In Section 4 we compare the centralized and distributed genetic algorithms with parallel versions of genetic algorithms introduced by Grefenstette [13]. In Section 5, we provide experimental results of centralized and distributed genetic algorithms for the *traveling salesman problem* and show that their performance is as good as standard genetic algorithms but that they run faster due to the speed-up obtained from the parallelism. Finally, in Section 6, we draw some conclusions and discuss some directions of future research.

2. Centralized Genetic Algorithms

In this section, we present an algorithm which parallelizes the standard genetic algorithm shown in Figure 1 and comment on some of its shortcomings.

Consider a pool of (identical) processors (called *slave processors*) which each contain a structure of the population and which can evaluate structures and perform recombination operations, such as cross-over, mutation or local improvement, and consider another processor (called the *master processor*) whose task it is to instigate and synchronize the evaluation and recombination steps and to perform the selection step. In Figure 2 we show the code executed by a slave processor and the code executed by the master processor.

```
                    SLAVE PROCESSOR

{
   if the master processor requests evaluation then
      evaluate the local structure;
   if the master processor requests recombination then
      perform recombination to the assigned structures;
}

                    MASTER PROCESSOR

{
   while termination condition does not hold
      {
         while any slave is active WAIT;
         perform (global) selection (this involves
          reassigning structures to slave processors);
         request evaluation from the slave processors;
         request recombination from the slave processors on
          assigned structures;
      }
}
```

Figure 2. A Centralized Genetic Algorithm.

As can be seen, parallelization and the accompanying speed-up is achieved by distributing the work required in the evaluation and recombination step over the slave processors. Notice, however, that the selection step is central to the task of the master processor. In fact, stated from a different perspective, it is because of the selection step that a master processor is necessary. This

is the case because selection, as described in the current literature, is a global process requiring knowledge about the values of all structures of the current population. It is for this reason that we call this algorithm a *centralized genetic algorithm.* Although, this algorithm is a natural parallel implementation of a standard genetic algorithm and achieves the speed-up it is designed for, it has disadvantages:

i. *the algorithm is not reliable*: indeed, as can be seen from the code shown in Figure 2, if the master processor or one of the slave processors fails, the algorithm halts.

ii. *synchronization delays may occur* because evaluation and recombination operations may not all require the same time when applied to different structures and therefore processor time is wasted in the form of idle time.

iii. *the algorithm does not appear natural because selection is centralized.* It seems to us that, in a broader sense, selection is a process that should not be centralized to a single processor; it certainly is not implemented as such in nature. In fact, in nature, selection, as a global effect, is achieved through the continuous interaction and competition of individual structures and is not controlled by a central agent. In Section 3 we will see how to overcome this very problem and obtain a more natural and uniform parallel genetic algorithm. It should also be noted that a slow-down of the algorithms can be expected because selection is not parallelized as opposed to evaluation and recombination.

3. Distributed Genetic Algorithm

In this section, we propose a framework in which the principal components of genetic algorithms can be implemented as local processes. We then argue why we think this implementation is more natural and at least as efficient as the standard implementation. Our framework consists of a pool of processors which execute identical or nearly identical tasks in parallel. Each processor has a local memory large enough to store a small number of structures, one of which will be called the *local structure.* The collection of all these local structures in the processors constitutes the population of the genetic algorithm, hence rather than a having a global memory to store the structures of the population, the structure are spread out over the processors in the form of local structures of processors. Furthermore, each processor is capable of performing local tasks and communicating with the other processors. In this framework, we can describe a new way of implementing genetic algorithms. As indicated before (see Figure 1), a genetic algorithm breaks down into the repeated application of an evaluation step, a recombination step and a selection step. It is straightforward to implement the evaluation and recombination steps. In the evaluation step, each processor evaluates its local structure and stores the outcome in its local memory. The recombination step which usually consists of a cross-over step and a local improvement step is implemented as follows:

i. For the cross-over step, each processor p elects to communicate with another processor q, with some locally controlled probability. After communication is established, processor p reads in the local structure of q, after which communication between p and q ceases to exists. Processor now performs cross-over between the structure just read in and its local structure and one of the offsprings becomes the new local structure of p.

ii. For the local improvement step, each processor probabilistically determines to perform local improvement on its local structure.

The novelty of our approach comes from the fact that we also propose to implement the *selection* step by local processes. In the standard genetic algorithm, the selection step is implemented by a single process which gathers the performance value of the structures, computes their average and "duplicates" the structures according to their relative performance with that average. If one wants to faithfully replicate this process, one has to introduce a special processor for this step of the genetic algorithm. In our opinion, this is unnatural as well as unnecessary. It is unnatural

since we do not believe in a supervising agent which, for each structure, assigns its rating and calculates the number of offsprings (certainly, nature does not seem to behave that way). It is also unnecessary since selection can be implemented, as will be seen shortly, by local processes. There are several ways to implement a selection step using local processes. What is common to all of them, though, is that they all implement a notion of the survival of the fittest principle. We next outline five different, but related, selection steps, called *Selection 1, Selection 2, Selection 3, Selection 4 and Selection 5.*

In Selection 1, each processor p, with some locally controlled probability elects to communicate with another processor q, if the value of the local structure of p is better than the value of the local structure of q, processor p overwrites the local structure of q with its local structure after which communication is ceased, otherwise p undertakes no action and communication is ceased immediately (notice that processor q is passive in this process). In Selection 2, each processor p, with some locally controlled probability elects to communicate (not necessarily simultaneously) with k other processors $q_i (1 \leq i \leq k)$, p reads in the value v_i of the local structure of q_i and stores the processor number of q_i and ceases communication with q_i. Processor p computes $av = 1/k \sum_{i=1}^{k} v_i$, the average value of the v_i's, and compares the value v of its local structure with av. If $v > av$ then p randomly selects another processor q and overwrites the local structure of q with its local structure, otherwise p undertakes no further action. In Selection 3, the following action is undertaken by processor p: if $v > av$ then p randomly selects one of the processors q_i (remember p has the processor numbers of the q_i's in its local memory) and overwrites the local structure of q_i, otherwise p undertakes no further action. In Selection 4, the following action is undertaken by processor p: if $v > av$ then p overwrites the local structure of the processor among the k processors q_i with the worst v_i-value, otherwise p undertakes no further action. In Selection 5, the following action is undertaken by processor p, if $v > av$ then p overwrites the local structures of l processors, with $l = \lceil min(k, v/av) \rceil$, of the selected processors q_i, otherwise p undertakes no further action. It is interesting to notice that Selection 1 is a special case of Selection 4 and Selection 5 for $k = 1$ and that Selection 5 is quite related to the standard selection step of sequential genetic algorithms. It should also be noted that we can easily incorporate normalization techniques as suggested by Baker [1] within these selection schemes.

In Figure 3 we summarize the above discussion by showing the code each processor executes during the course of a run of a distributed genetic algorithm. Notice that we do not require that a processor executes the four statements in the while loop in the specified order or that p_l, p_c or p_s are the same for all processors.

```
{
   while termination condition does not hold
      {
         evaluate the local structure;
         perform local improvement on the
          local structure with probability p_l;
         perform cross-over with probability p_c;
         perform local selection with probability p_s;
      }
}
```

Figure 3. A Typical Processor of a Distributed Genetic Algorithm.

We are now ready to give a description of a distributed genetic algorithm. A *distributed genetic algorithm* (DGA) consists of a pool of processors as described above which are initialized by assigning to each of them a local structure and are then run asynchronously with each processor

executing its local code as shown in Figure 3. The DGA adopts the following synchronization policy: if a processor p wants to communicate with another processor q, p places a lock on q which is released when communication between the two processors ceases; if during this communication another processor r wants to communicate with q, communication between r and q is not granted and r proceeds by trying to communicate with another processor. Notice that this simple synchronization policy can be implemented by local processes as well.

There are certain observations we want to make about distributed genetic algorithms:

i. *they yield similar performance as standard genetic algorithms*: experiments with DGAs on function optimization problems and combinatorial optimization problems yielded performance, both in speed and in robustness, of the same quality as experimental results with the standard sequential genetic algorithms reported in the literature. In Section 4, we compare the performance of a DGA and a standard genetic algorithm for the traveling salesman problem.

ii. *their implementation is straightforward and uniform due to the introduction of the local selection process*: in standard genetic algorithms, the selection step is a globally controlled process. This results in an asymmetry in their implementation since selection has to be considered separately from the evaluation step and the recombination step. In the distributed version, this asymmetry is removed and uniformity is obtained by localizing the selection step and therefore localizing all major components of the genetic algorithm. The global effect of a genetic algorithm is obtained because the processors communicate when performing crossover and selection. From an implementation point of view, also notice that we do not need sophisticated locking and scheduling mechanisms and that there is only a minimal contention problem [22] since processors rarely will be competing for the same memory locations.

iii. *their implementation is more natural*: in our opinion, the distributed genetic algorithm resembles closer the evolutionary process found in nature. In nature a pool of structures communicate and operate on each other in the form of local processes to yield the effect known as the evolutionary process. It does not appear likely that there is a supervising agent which controls this process or even parts of this process such as the selection step. In fact, we strongly believe that selection in nature is achieved through local processes which perform a kind of survival of the fittest strategy.

iv. *they are very reliable algorithms*: the failure of a processor only slightly alters the flow of the algorithm, in the worst case, the processor that fails has a lock on another processor and therefore disables that processor upon failure, but this will not have a major effect on the communication and the actions of the other processors, resulting in only a minor change in the flow of the entire algorithm. It should also be noticed that is is very easy to repair or insert processors without affecting the algorithm much.

v. *they allow for effects not possible in synchronized implementations of genetic algorithms*: due to the asynchronous behavior of the algorithm, different processors may display different behaviors. For example assume we have a processor g with a "good" local structure and a processor b with a "bad" local structure. It is likely that processor b will spend more time improving its structure by performing local improvements on its local structure, whereas processor g may in the mean time spend his time communicating with other processors through crossover or selection. Clearly this effect is by-passed in synchronized implementations of genetic algorithms such as the centralized genetic algorithms.

vi. *they offer more tuning opportunities*: since the crossover probability p_c, the local improvement probability p_l, the selection probability p_s, as well as the actual crossover, local improvement and selection routines are local to each individual processor (in contrast, in the standard genetic algorithms all these parameters and routines are the same) a DGA allows for more tuning opportunities by setting these parameters and operators not necessarily equal in all the

processors. It is, for example, quite likely that the parameters should change over the course of the algorithm and may change according to the properties of the local structure of each processor. This ability offers, for example, additional techniques to overcome the premature convergence problem found in most genetic algorithms. In fact, we have already observed this phenomenon in our experiments with DGAs.

4. Comparison with Other Parallel Implementations of Genetic Algorithms

There have been other proposals to parallelize genetic algorithms, the most noticeable among these, the proposal of Grefenstette [13]. We will state Grefenstette's assumptions, give two of his parallel genetic algorithms and along the way, compare and contrast his approach with ours.

Grefenstette's main assumption is that the dominant cost in a genetic algorithm is the amount of time spent in doing function evaluations. In other words, he assumes that the evaluation step takes the most time and the recombination and selection steps are merely small overhead. While this is a reasonable assumption in some applications, such as optimizations of simulations [12] and evolving production system programs for AI tasks [24], this assumptions is not valid in other applications such as some combinatorial optimizations problems like the traveling salesman problem or puzzle problems such as the sliding puzzle problem [25], where in fact as much time or even more time is spent in the recombination step as in the evaluation step due to the incorporation of heuristics in the crossover and local improvement operators [14, 15, 25]. Given Grefenstette's assumption, it is difficult to compare his algorithms with the distributed genetic algorithm, but it is still interesting to contrast both approaches. We state two of his algorithms next. *Algorithm 1* is an algorithm with a centralized concurrency control mechanism (we show this algorithm in Figure 4). Much like the CGA described in Section 3 it consist of $k + 1$ processors, one master processor and k slave processors. The master process maintains the population of structures and performs the selection and recombination steps of the genetic algorithm. The slave processors are responsible for structure evaluation.

Comparing Algorithm 1 and the CGA is left up to the reader. Algorithm 1 and the DGA basically coincide in that they both distribute function evaluations but greatly differ in the way recombination and selection is performed. In Algorithm 1, the master processor is responsible for these processes, in the DGA, recombination and selection are distributed in the form of local processes. As mentioned by Grefenstette, Algorithm 1 has rather poor reliability characteristics. If the master process fails, the entire algorithm halts. Furthermore, the synchronization mechanism employed relies on the fact that all slave processors successfully complete their actions. As mentioned in Section 3, the DGA is highly reliable, i.e., failure of a processor does only marginally affect the performance of the entire algorithm.

Algorithm 2 uses distributed, asynchronous concurrency control. There are k identical processors, one of them is shown Figure 5.

Although this algorithm is closer to a DGA since the evaluation and the recombination steps are distributed, it differs from a DGA in two ways:

i. selection is not localized since each processor has to update the selection probabilities of all structures of the population, and

ii. Algorithm 2 does not distribute memory, instead there is one global memory which stores the structures of the population. This can lead to contention problems as indicated by Grefenstette and can thus result in a slow-down of the algorithm. In contrast, in the DGA, there is no notion of a global memory which stores the population of structures, rather, the pool of processors with their local structures serves in the role of the population of the genetic algorithm. As indicated in Section 3, this implies that a simple locking mechanism with little contention problem suffices to implement successful communication, resulting in maximal speed-up given

```
                    SLAVE PROCESSOR

{
   while there are unevaluated structures in the population
      {
         choose a subset s_j1,...,s_jn
          of size n (where n = (size of the population)/k)
          from the set of unevaluated structures
          in the population;
         evaluate each of the chosen structures;
      }
}

                    MASTER PROCESSOR

{
   while termination condition does not hold
      {
         while any slave is active WAIT;
         perform the (global) selection step;
         perform the recombination step;
      }
}
```

Figure 4. Algorithm 1 of Grefenstette.

```
{
   while termination condition does not hold
      {
         remove n unevaluated structures from the population;
         evaluate the chosen structures;
         recombine the chosen structures;
         { enter critical section
            insert the structures into the population;
            update the selection probabilities;
         } leave critical section
      }
}
```

Figure 5. Algorithm 2 of Grefenstette.

an implementation on a multi-processor machine.

5. Experimental Results with Centralized and Distributed Genetic Algorithms

In this section, we present two sets of experimental results about centralized and distributed genetic algorithms. In the first set, we compare the results of a simulation of a CGA and a DGA on a sequential machine, a VAX 8800. In the second set, we compare the results of implementations of both algorithms on a multi-processor machine, a Butterfly machine with 16 processors [2].

The algorithms are applied to the Traveling Salesman Problem (TSP) [19, 20]. Those who are not familiar with this application, we refer to [11, 14, 15, 25] where standard genetic algorithms

are described to (approximately) solve this problem. Before discussing the results in detail, we would like to point out that genetic algorithms for the TSP do not satisfy the central assumption of Grefenstette [13]. As mentioned in Section 4, Grefenstette assumed that the evaluation of structure takes more time than the recombination operators. But, in case of genetic algorithms for the TSP, the recombination operators, i.e., the cross-over and local improvement operators, take as much as or more time than the evaluation.

5.1 Simulation Results on a Sequential Machine

Three different TSP problems were analyzed, their names and definitions are shown in Figure 6.

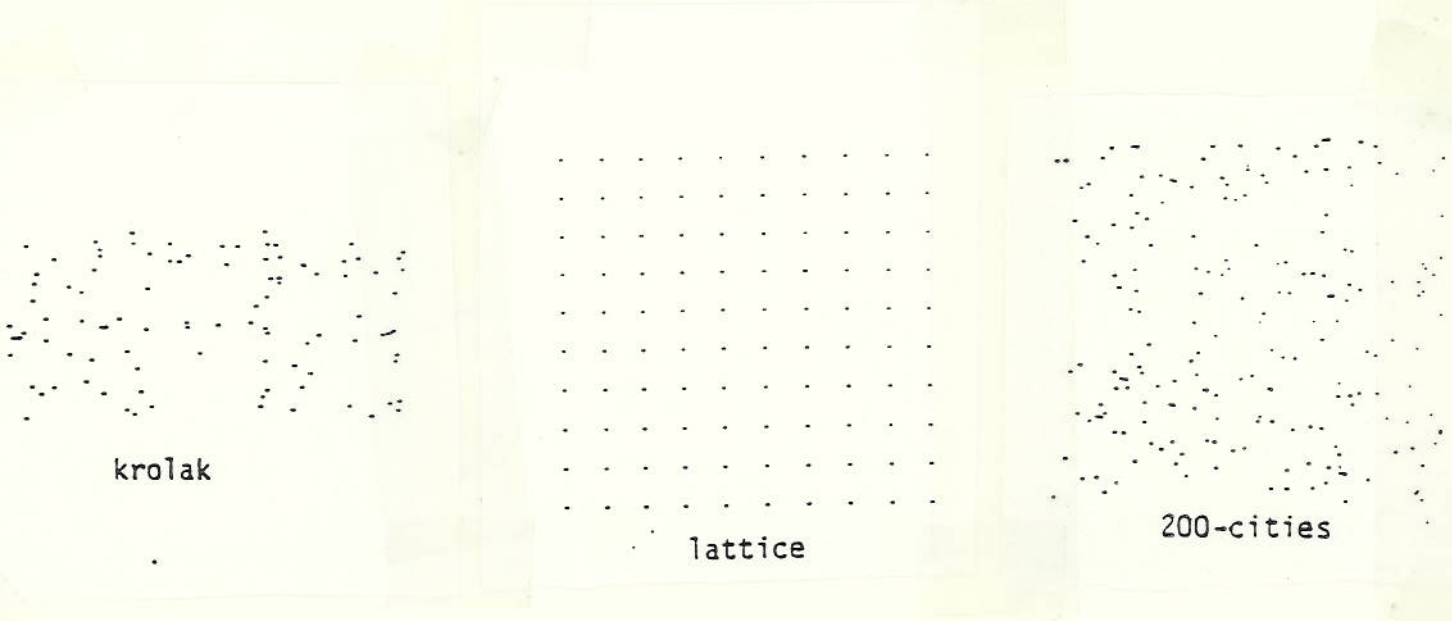


Figure 6. Three Traveling Salesman Problems.

The simulation of the CGA corresponds exactly to the standard GA. In the simulation of the DGA, we used Selection 1 as the local selection procedure, i.e., if processor p (with a certain probability p_s) elects to communicate with a processor q then if the value of the local structure of p is better than the value of the local structure of q, processor p overwrites the local structure of q with its local structure.

In Appendix 1 we show the results of running the simulations of the above described DGA and CGA. The most important observation is that the performance of the DGA which uses a local selection method is similar to that of the CGA which uses the standard selection method. This result indicates that it is possible to safely use the more natural distributed genetic algorithms and still obtain similar results. If there was a difference in the performance of the two algorithms, it was in the fact that local selection seems to add another source of maintaining the diversity because of its more noisy behavior. As indicated in previous work, this can only add to the robustness of the algorithm. In fact the noisyness of the local selection allowed us to use fewer local improvements (mutations) than was necessary for the CGA.

5.2 Some Parallel Implementation Results

The experiments described in this section were done on a Butterfly [2] machine with 14 available processors. Only one of three traveling salesman problems, the Lattice problem with 20, 60, and 100 cities was looked at in these experiments (we expect similar results for other TSP problems).

We ran each algorithm with a varying number of processors. This requires some comments because our algorithms are originally designed in such a way that one processor holds only one structure. What we mean by this is that, if we run for example a DGA with 100 structures on 2 processors, we allocate 50 structures on each processor and one processor sequentially simulate the task of 50 processors. So even in case of 14 processors running, each processor has to simulate 7 or 8 processes sequentially. The ideal case would be to run 100 processors in parallel. The purpose of running the algorithm on a varying number of processors is to find out how far the algorithm is parallelizable by deriving the curve of "effective processors". In most cases, using k processors does not result in k-times speed-up due to the overhead of communication or a section of program which cannot be parallelized. If we run a program with a single processor and then run it with k processors, we can find out how much speed-up k processors produced, by dividing the execution time of single processor run by that of k processors. For each k, we can derive the speed-up factor and draw a corresponding graph. This curve is usually a convex (sublinear) curve. The less convex it is, the better. In our actual experiments, we could not calculate these curves as stated because we were unable to run the program on a single processor because of its size. Instead, we calculated the speed-up factor against the running time of 2 processors. The speed-up curves for the experiments using the CGA and the DGA are shown in Appendix 2. As can be seen, these curves are not smooth. This is due to the fact that for experiments with different number of processor, different random seeds have to be used in the different processors, resulting in a slightly different behavior of the algorithm. In the case of the CGA, the speed-up is better as the size of problem increases. The reason is that the overhead the CGA carries, due to the global selection and the synchronization delays of the master processor, is sufficient to slow down the speed-up in this small size problem. In the case of the DGA, there is no global selection and fewer synchronization delays. So the speed-up curves show less change as the size of the problem increases although it still appears that speed-up is better in larger size problem. We would like to note that as the local selection we used is a variant of Selection 2: we take 5 samples and when we overwrite a structure, we make sure that it is overwritten by a better structure, if the one to be overwritten is better, no overwriting occurs. As local improvement method we used simulated annealing [3, 4, 18, 25] where the initial temperature is chosen to be some fraction of the standard deviation of the values of the population and this temperature is decreased exponentially. Also we used a stopping mechanism which is different from the standard stopping mechanism which consists in just counting number of evaluations. A more detailed version of this DGA is shown in Appendix 3.

6. Conclusion

We have shown that genetic algorithms can be modified to become distributed asynchronous algorithms, which we called distributed genetic algorithms. This is done by localizing the selection step and distributing it together with the evaluation and recombination steps to a pool of processors. Our experiments indicate that the solutions obtained by DGAs are as good as the ones obtained by the standard genetic algorithms which we implemented as centralized genetic algorithms. The advantage of DGAs are that they are reliable, natural algorithms which, when implemented on a parallel machine, can result in very fast search algorithms. Other, more conventional, sequential search algorithm are much harder to parallelize. This point is well explained in [17]. In short, many such algorithms accumulate improvements on a single structure. This implies that one cannot easily break up those improvements into small pieces and use a pool of processors to perform them independently, i.e., it is usually the case that one piece of improvement has to be done first in order for another piece of improvement to become effective. Distributed genetic algorithms do not suffer from such complications. Despite the contention problems (which are minimal), speed-up factors can be expected to be much higher than speed-up factors obtained for other

search algorithms, for example [9]. It would be quite worthwhile to conduct an empirical study comparing the performance of DGAs with that of parallelized versions of other search algorithms.

References

[1] J. Baker," Adaptive Selection Methods for Genetic Algorithms", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 101-111 (July 1985).

[2] BBN Advanced Computers. The Uniform System approach to programming the Butterfly parallel processor. Rep. 6149, Version 2.

[3] E. Bonomi, Jean-Luc Lutton, " The N-city Traveling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm " SIAM Review Vol. 26 No. 4 October 1984 pp 551-568

[4] V. Černý, " Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm" Journal of Optimization Theory and Application Vol.45 No. 1 January 1985 pp 41-52

[5] L. Davis, "Job Shop Scheduling with Genetic Algorithms", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 136-140 (July 1985).

[6] L. Davis, "Applying Adaptive Algorithms to Epistatic Domains", Proc. of 9th IJCAI, pp. 162-164 (Aug 1985).

[7] K.A. De Jong, "Adaptive System Design: a Genetic Approach", *IEEE Trans. Syst., and Cyber. Vol. SMC-10(9)*, pp. 556-574 (September 1980).

[8] K.A. De Jong, "Genetic Algorithms: a 10 Year Perspective", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 169-177 (July 1985).

[9] Raphael A. Finkel, John P. Fishburn " Parallelism in Alpha-Beta Search" Artificial Intelligence 19(1) September 1982, pp 89-106

[10] J.M. Fitzpatrick, J.J. Grefenstette and D. Van Gucht, " Image Registration by Genetic Search", Proceedings of IEEE Southeastern April 1984, pp 460-464

[11] D.E. Goldberg and R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 154-159 (July 1985).

[12] J. J. Grefenstette, "Optimization of Control Parameters for Genetic Algorithms", IEEE Trans. Systems, Man, and Cybernetics (1985)

[13] J.J. Grefenstette, "Parallel Adaptive Algorithm for Function Optimization" Technical Report CS-81-9 Computer Science Dept. Vanderbilt University November 1981

[14] J.J. Grefenstette, R. Gopal, B.J. Rosmaita and D. Van Gucht, "Genetic Algorithms for the Traveling Salesman Problem", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 160-168 (July 1985).

[15] J. J. Grefenstette, "Incorporating Problem Specific Knowledge into Genetic Algorithms", To appear

[16] J. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor (1975).

[17] P. Jog, D. Van Gucht, "Parallelization of Probabilistic Sequential Search Algorithm", Technical Report, Indiana University, April 1987 To appear in the Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications.

[18] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing", *Science Vol. 220*(4598), pp. 671-680 (May 1983).

[19] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (Ed), *The Traveling Salesman Problem*, John Wiley & Sons Ltd (1985).

[20] S. Lin and B.W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem", *Operations Research* 1972, pp. 498-516.

[21] M. L. Maudlin, " Maintaining Diversity in Genetic Search", *Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications*, pp. 247-250 (July 1985).
[22] R. Rettberg, R. Thomas "Contention is No Obstacle to Shared-Memory Multiprocessing", *CACM* December 1986, pp. 1202-1212
[23] D. Smith, "Bin Packing With Adaptive Search", Proc. of an Int'l Conference on Genetic Algorithms and Their Applications, pp. 202-206 (July 1985).
[24] S.F. Smith, "Flexible Learning of Problem Solving Heuristics Through Adaptive Search", Proc. of 8th IJCAI (Aug. 1983).
[25] J. Y. Suh, D. Van Gucht, "Incorporating Heuristic Information into Genetic Search", Technical Report, Indiana University, February 1987 To appear in the Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications.

APPENDIX 1

** Illustration on Parameters and Tables of Statistics **
** On TSP Experiment. **

Population Size : the number of population on given time.
Structure Length : the length of structure (in this case, the number of cities).
Crossover Rate : the portion of population undergoing cross-over. the rest will undergo the local improvement.
Local Rate : if you mutiply this ratio to the structure length, you will get the number of local improvement attempts to be done for a structure.
Selection Rate : the probability that each strucuture will undergo the LOCAL SELECTION.

** How to read a table **

Population Size = 100
Structure Length = 100
Crossover Rate = 0.5
Selection Rate = 0.5

Algorithm		DGA	CGA
Local Rate		5 %	10 %
lattice	exp. 1	101.6 (139:14000)	100 (188:16910)
	exp. 2	101.6 (149:15000)	100.8 (200:17786)
	exp. 3	100.8 (179:18000)	100 (207:18865)
	exp. 4	100.8 (139:14000)	100 (237:22538)
	exp. 5	102.0 (189:19000)	100.8 (163:13650)

First 3 parameters above the table are those of both DGA and CGA. The last one "Selection Rate" is that of DGA. The above table is for TSP experiment for Lattice problem. We conducted 5 experiments for two different algorithms. The first row shows the result of DGA with Local rate 5 %. The description "exp. i" means experiment i. nnn.n (ggg: tttt) means that nnn.n is the best solution after ggg generations which took tttt trials. ggg and tttt for DGA are approximate values.

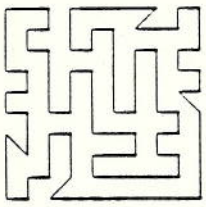
** In what follows we show the results obtained for the three TSP pro-
** blems shown in Figure 2.

1. LATTICE problem

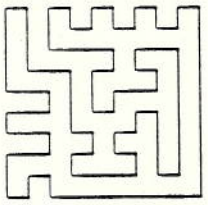
Population Size = 100
Structure Length = 100
Crossover Rate = 0.5
Selection Rate = 0.5

Algorithm		DGA	CGA
Local Rate		5 %	10 %
lattice	exp. 1	101.6 (139:14000)	100 (188:16910)
	exp. 2	101.6 (149:15000)	100.8 (200:17786)
	exp. 3	100.8 (179:18000)	100 (207:18865)
	exp. 4	100.8 (139:14000)	100 (237:22538)
	exp. 5	102.0 (189:19000)	100.8 (163:13650)

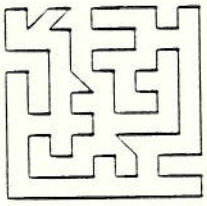
exp 1: 101.6



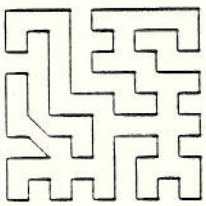
exp 1: 100.0



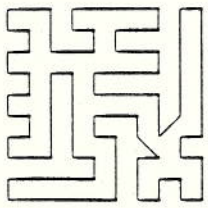
exp 2: 101.6



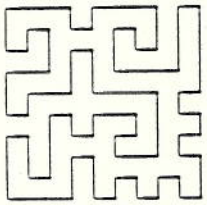
exp 2: 100.8



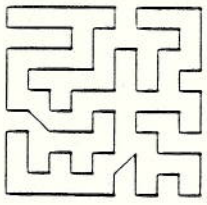
exp 3: 100.8



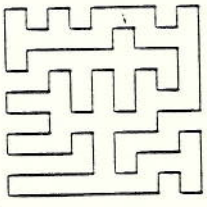
exp 3: 100.0



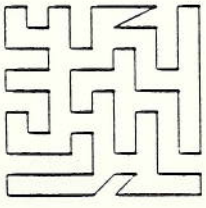
exp 4: 100.8



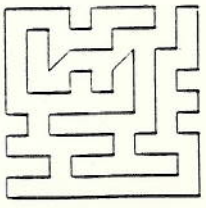
exp 4: 100.0



exp 5: 102.0



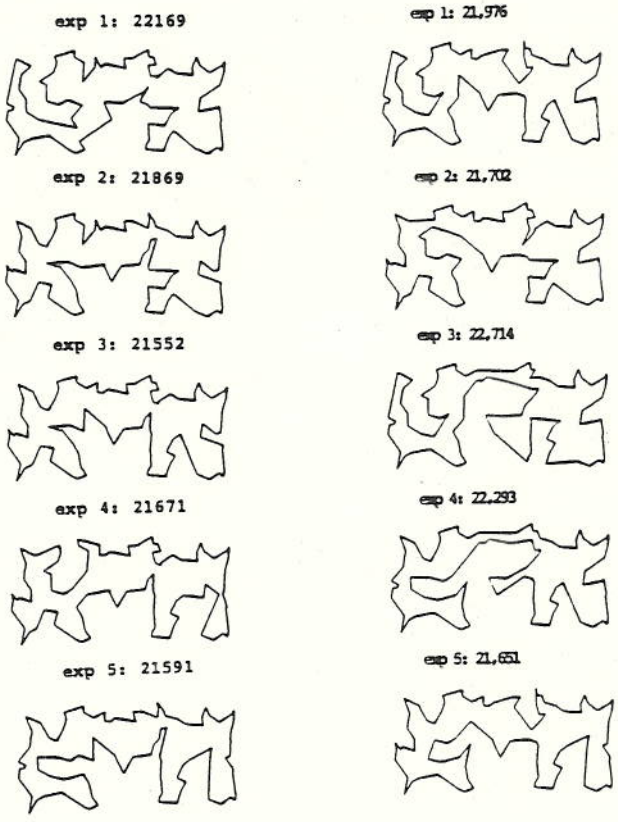
exp 5: 100.8



2. KROLAK problem

Population Size = 100
Structure Length = 100
Crossover Rate = 0.5
Selection Rate = 0.5

Algorithm		DGA	CGA
Local Rate		5 %	10 %
krolak	exp. 1	22169 (319:32000)	22293 (373:29435)
	exp. 2	21869 (489:49000)	22714 (386:30361)
	exp. 3	21552 (369:37000)	21702 (403:31536)
	exp. 4	21671 (259:26000)	21976 (627:49482)
	exp. 5	21591 (409:41000)	21651 (679:49745)

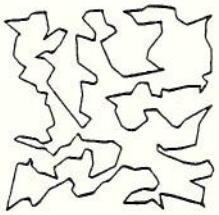


3. 200 cities

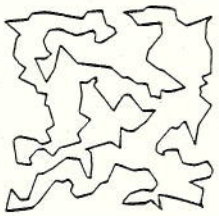
Population Size = 100
Structure Length = 200
Crossover Rate = 0.5
Selection Rate = 0.5

Algorithm		DGA	CGA
Local Rate		5 %	10 %
200	exp. 1	160.2 (939:94000)	154.8 (679:52312)
	exp. 2	154.6 (719:72000)	158.4 (999:78890)
	exp. 3	153.8 (919:92000)	158.0 (711:57030)
	exp. 4	152.9 (699:70000)	158.5 (768:60148)
	exp. 5	155.5 (679:68000)	153.6 (946:72553)

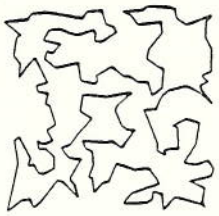
exp 1: 160.25



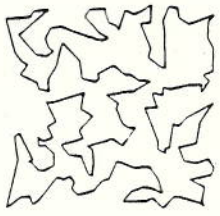
exp 1: 154.87



exp 2: 154.6

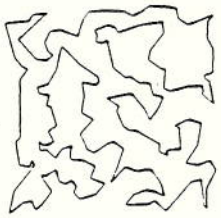


exp 2: 158.44



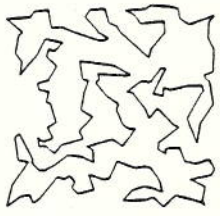
exp 3: 153.8

exp 3: 158.08



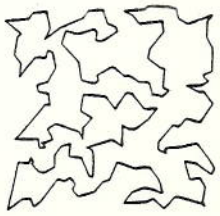
exp 4: 152.9

exp 4: 158.55



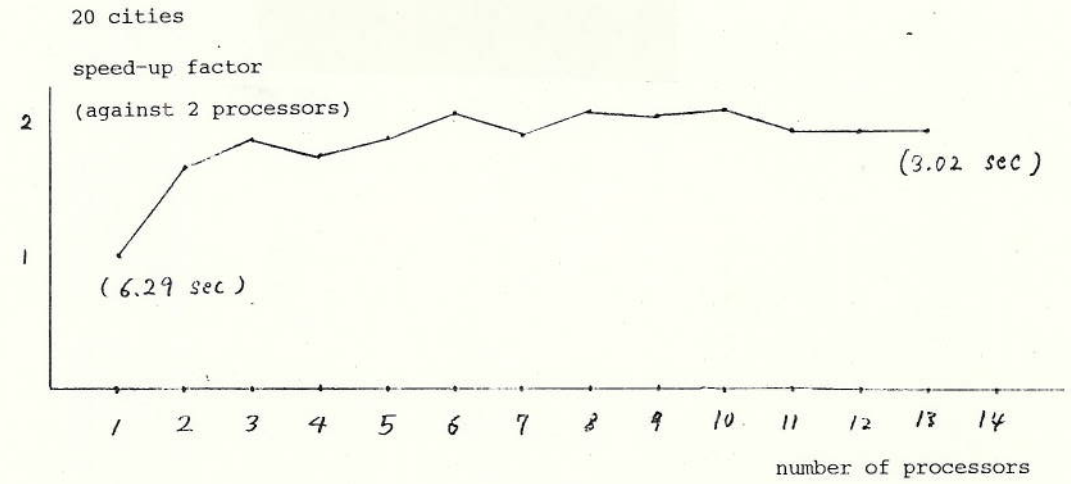
exp 5: 155.5

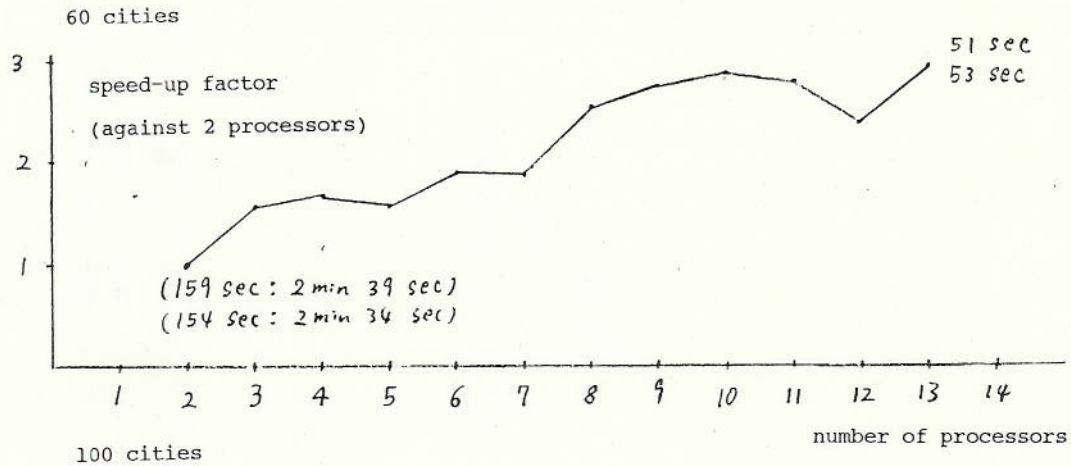
exp 5: 153.67

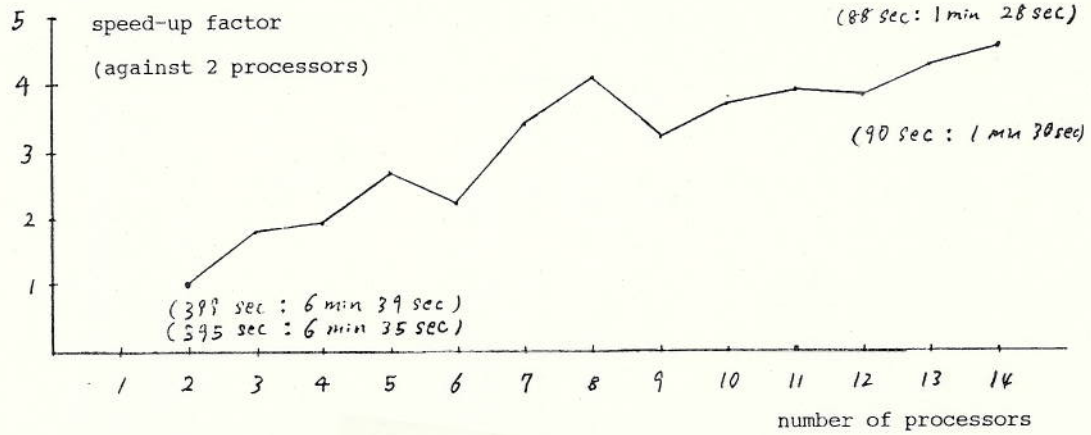


APPENDIX 2

Speed-up Curve of CGA

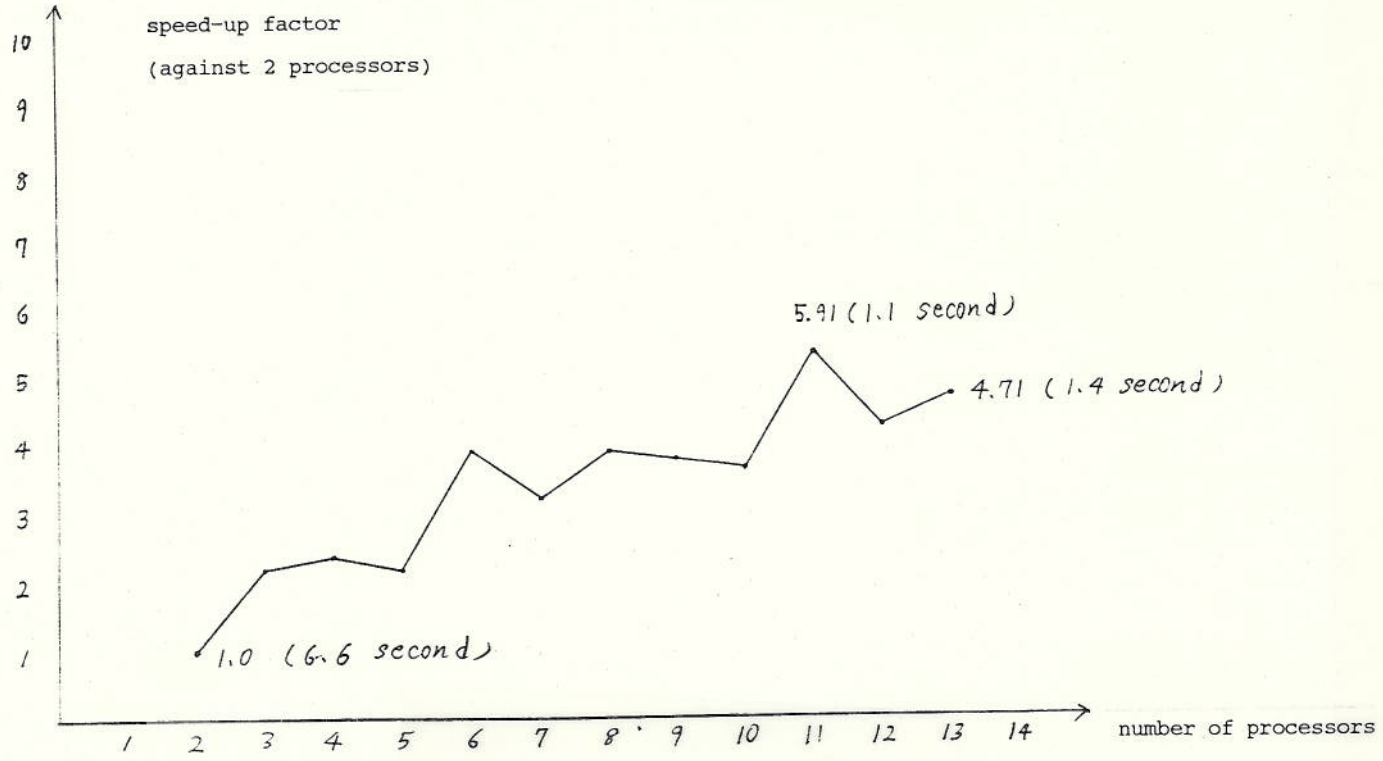




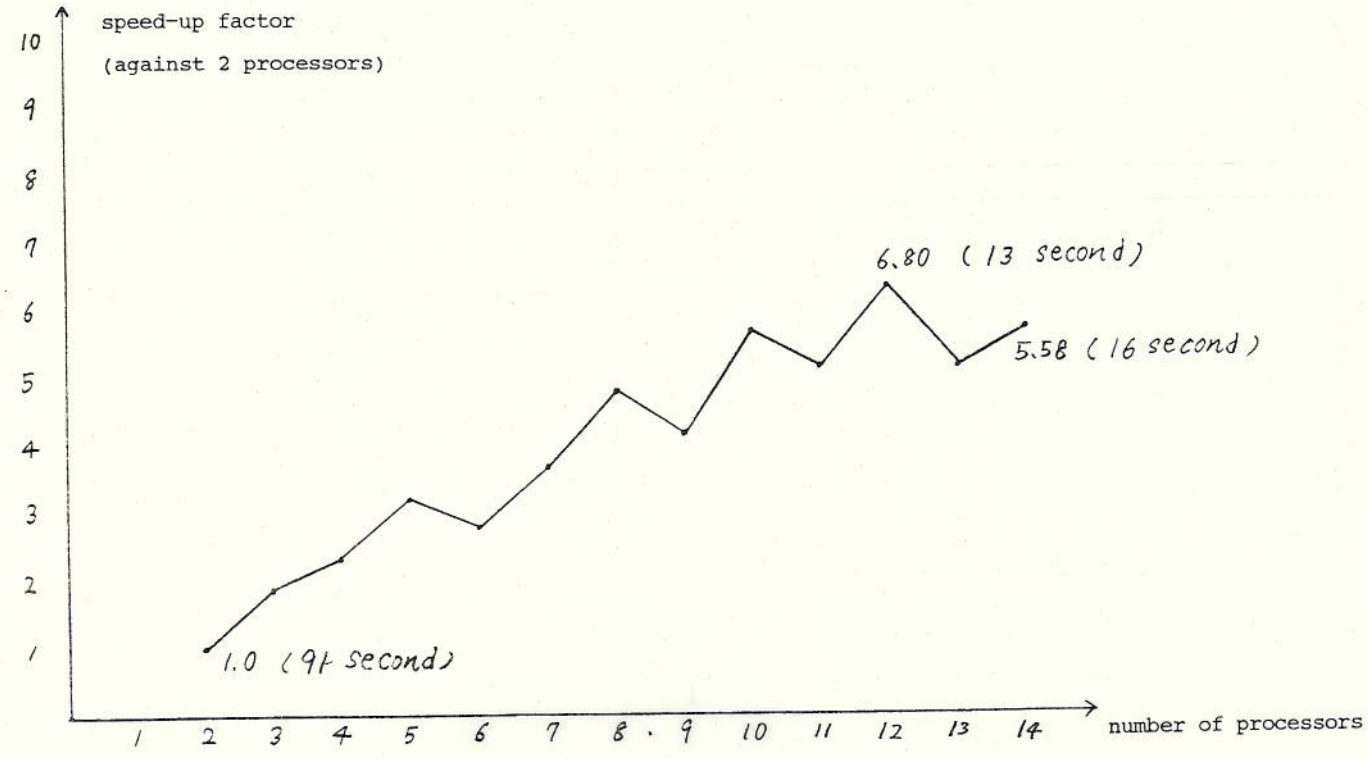


Speed-up Curve of DGA

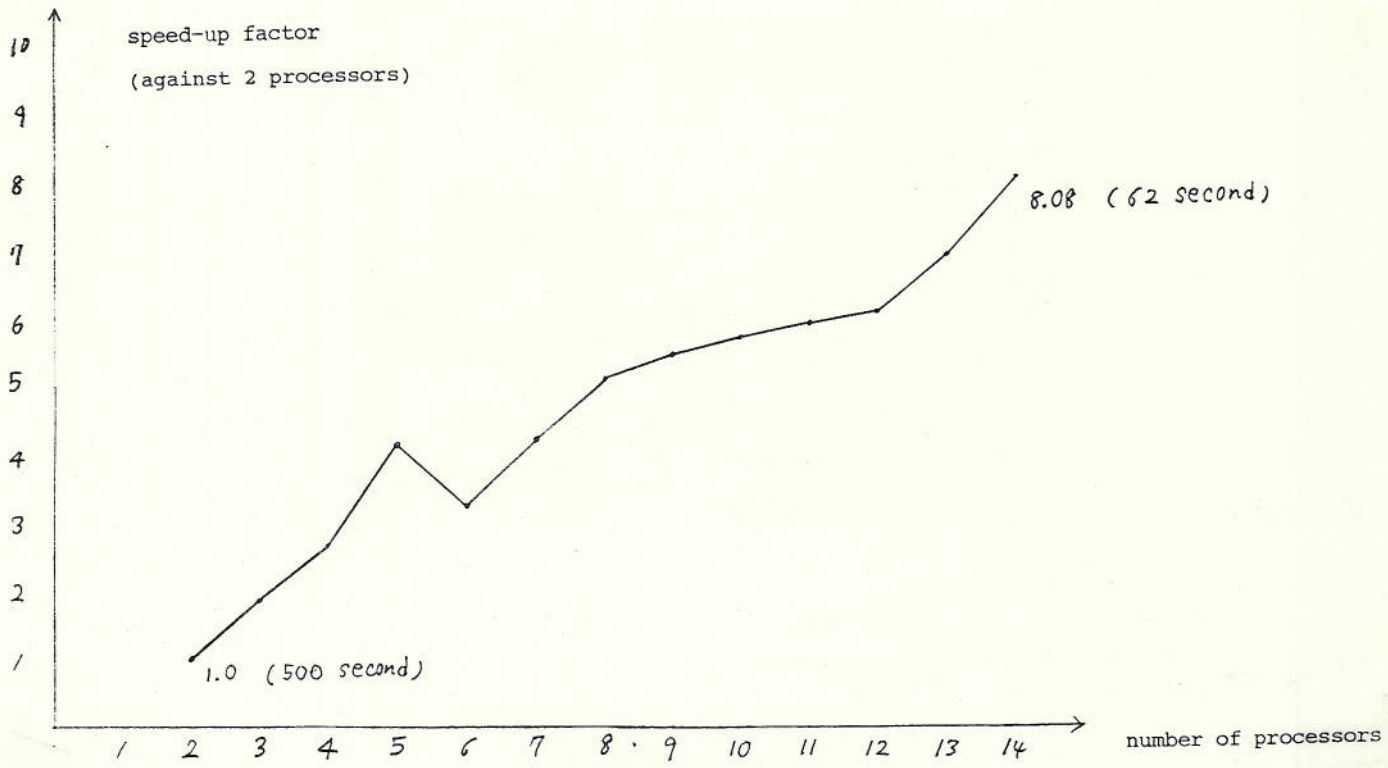
20 cities



60 cities



100 cities



List of Solutions of CGA

1. 20 cities

Optimal Solution : 20
Population Size : 20
Processors used : 14

best solution	experiment 1	21.23
	experiment 2	20.00
	experiment 3	20.00
	experiment 4	21.23
	experiment 5	20.00

2. 60 cities

Optimal Solution : 60
Population Size : 60
Processors used : 14

best solution	experiment 1	60.00
	experiment 2	61.65
	experiment 3	60.82
	experiment 4	60.82
	experiment 5	61.65

3. 100 cities

Optimal Solution : 100
Population Size : 100
Processors used : 14

best solution	experiment 1	100.82
	experiment 2	100.82
	experiment 3	100.82
	experiment 4	100.00
	experiment 5	100.00

List of Solutions of DGA

1. 20 cities

Optimal Solution : 20
Population Size : 20
Processors used : 14

best solution	experiment 1	20.00
	experiment 2	20.00
	experiment 3	20.00
	experiment 4	20.00
	experiment 5	20.00

2. 60 cities

Optimal Solution : 60
Population Size : 60
Processors used : 14

best solution	experiment 1	61.99
	experiment 2	61.65
	experiment 3	61.99
	experiment 4	60.82
	experiment 5	61.41

3. 100 cities

Optimal Solution : 100
Population Size : 100
Processors used : 14

best solution	experiment 1	101.41
	experiment 2	101.65
	experiment 3	100.82
	experiment 4	100.82
	experiment 5	100.82

APPENDIX 3

Detailed Description of the DGA

Parameters

l : the number of cities.
m : the size of the population (equivalently m = number of processors)
P : the population.
$\mathbf{t} = (t_0, t_1, ..., t_{m-1})$: array of private clocks of the processors.
Each initially starts with 0.
$\mathbf{conv} = (conv_0, conv_1, ..., conv_{m-1})$: array of private counters,
used for determining the convergence of the algorithm.
Each initially starts with 0.
C : the set of all possible tours.
R^+ : the set of positive real numbers.
f : the function of C to R^+, which returns the performance measure of tour, i.e., the length of tour.
p_c, p_s, p_l: the probability that a tour will undergo cross-over, local improvement (simulated annealing), and local selection respectively.

Setting

a. There are m processors, $proc_0, proc_1, ..., proc_{m-1}$. Each processor $proc_i$ has one local tour c_i in its local memory.

b. c_i is stored in an array cur_i. There is another array new_i in the local memory which is used to hold a new tour derived by **cross-over** or **local improvement (simulated annealing)**.
There are two additional arrays, mom_i and dad_i. They are needed to avoid the problem of structures being overwritten while they are being used for other operations.

c. There are other local variables cv_i which holds $f(cur_i)$.
nv_i is the same kind of local variable which will store $f(new_i)$, if necessary.
In the same fashion, $momv_i$ and $dadv_i$ hold $f(mom_i)$ and $f(dad_i)$ respectively.

DGA()

begin

GenTaskEachProc (**Initialize**, 0); /* m processors run **Initialize** in parallel */

GenTaskForEachProc (**Operation**, 0); /* m processors run **Operation** in parallel */

/***** **OutputBest** () *****/

Output the best tour c^* and its length $f(c^*)$;

end

Initialize ()

begin

/* assume that the local memory has c_i */

Randomly generate a tour c_i and store it in cur_i; /* initialize a tour */
Compute $f(cur_i)$ and store it in cv_i; /* initialize the length of tour */
$t_i = 0$; /* initialize a local time step */
$conv_i = 0$; /* initialize a convergence counter */
end

Operation ()

begin

/* assume that the local memory has c_i */

Repeat
lock the array cur_i and its performance cv_i;
copy them into mom_i and $momv_i$;
unlock cur_i and cv_i;

with probability p_c:
begin
Randomly pick a tour c_j from cur_j in the local memory of $proc_j$;

lock the array cur_j and its performance cv_j;
copy them into dad_i and $dadv_i$;
unlock cur_j and cv_j;

Do **cross-over** with mom_i, dad_i to produce new_i;
compute $f(new_i)$ and store it into nv_i;

lock cur_i and cv_i;
swap cur_i, cv_i with new_i, nv_i; /* locally update the population */
unlock cur_i and cv_i;
end;

with probability p_s:
begin
Do **simulated annealing** with mom_i to produce new_i;

lock cur_i and cv_i;
swap cur_i, cv_i with new_i, nv_i; /* locally update the population */
unlock cur_i and cv_i;
end;

with probability p_l:
Do **local select**;

$t_i + +$;

Until (locally converged());

end

cross-over ()

The cross-over operator used here is the so-called *heuristic cross-over operator*. It is designed to ensure that the resulting configuration is a valid tour and a possibly better one. This kind of cross-over was introduced in [14]. It goes as follows:

note : In the below, an edge means a line connecting
two cities.
Pick a random city as the starting point for the child's tour. Compare the two edges leaving the starting city in the parents and choose the shorter edge.
Continue to extend the partial tour by choosing the shorter of the two edges in the parents which extend the tour. If the shorter parental edge would introduce a cycle into the partial tour, then extend the tour by a random edge. Continue until a complete tour is generated.
(This phrase is taken from [14])

simulated annealing ()

This operation replaces the random mutation operator used in previous genetic algorithms. As mentioned in [25], it plays a critical role in improving the efficiency of GA. It was inspired by the $2-opt$ operator of Lin and Kernighan [20]. Several researchers used it to devise an efficient algorithm for the TSP. Especially some of them saw it as a good way of generating a new tour from the existing one in the application of simulated annealing to the TSP [3,4,18]. Ours is a modified version of the version used by Černý [4] and basically is the following:

Pick randomly 2 edges e_0, e_1 where e_0
is from city x_0 to y_0 and e_1, from x_1 to y_1.
Make sure that all 4 cities are different.
Let d_0 be an edge from x_0 to y_1 and d_1, from x_1 to y_0.
Let the new tour τ be the modification of the old one where edges
d_0, d_1 replace e_0, e_1 respectively and those edges between
e_0 and e_1 are reversed.
Let $\Delta e = |e_0| + |e_1|$,
$\Delta d = |d_0| + |d_1|$
As can easily be seen,
the length of τ = (the length of old tour) + $\Delta d - \Delta e$;

If $(\Delta d - \Delta e < 0)$ /* new tour is shorter */
accept τ
Else
with probability $\exp((\Delta e - \Delta d)/T)$
accept τ where /* new tour is not shorter, but still acceptable */
$T = T_0 \rho^{t_i}$,
T_0: initial temperature,
ρ: cooling ratio, $(0 \leq \rho \leq 1)$
t_i: the local time step;
(T_0, ρ) is called **annealing schedule**

otherwise keep the old one.

Repeat the above operations a specified amount of times
/* This number of repetition should be specified by an user */

local select ()

begin

/* assume that c_i is in the local memory */

Randomly pick c_j in $proc_j$;
if $momv_i$ is better than cv_j then
begin
Randomly pick $j_0, ..., j_4$;
$\mu = \sum_{k=0}^{4} cv_{j_k}/5$; /* rough estimate of mean of population performance */
if $momv_i$ is better than μ then /* probably c_i is good enough */
begin
lock cur_j and cv_j;
$cur_j := dad_i$; /* overwrite c_j */
$cv_j := dadv_i$;
unlock cur_j and cv_j;
end
end
end

Note on local select

In this routine, good tours attempt to overwrite bad ones. In this way good tours are propagated and bad ones are eliminated. To achieve this, we need to ensure that a good tour is actually a good tour in the global context. That is, it should be good among those in the whole population. The criteria for a good tour is to be above the average tour length. Since it is out of the question to compute the population average (this would defeat the gains obtained by adopting local selection rather than global selection), we use, in this case, a sample average of five tours as an alternative. This is clearly the approximation of the average tour length. If, now, a structure is better than this sample average, we are almost certainly guaranteed that it is a good structure.

locally converged ()

begin

/* ϵ is an external constant which is quite small */

Randomly pick $j_0, ..., j_4$;
$\mu = \sum_{k=0}^{4} cv_{j_k}/5$;

/* does the population appear to be converged ? */

if $|\mu - cv_i| < \epsilon$ $conv_i++$;

if $conv_i > t_i/10$ return($true$);
else return($false$);
end

Rationale for using this stopping algorithm

In the CGA, the average performance is smoothly going down and the variance of population performance is almost monotonically decreasing. The population eventually converges and the time to get to the converged population stays more or less constant for a given instance of problem. In contrast, the DGA shows a more noisy behavior. The average performance does go down and the variance shrink, but they tend to fluctuate much more than those of the CGA. It is hardly exepected that the variance of performance

of population converges to 0, and even if it does, it takes an unreasonably long time. It is usually the case that the best structure is found well before the convergence occurs. There is another problem with the convergence pattern of DGA. Many times the population looks converged and no further improvement of population appears possible, only to diverge and a better structure emerges. That is, DGA often shows "false convergence". Thus it is not reasonable to use the convergence of the population as a criterion for stopping the DGA. We need to look for another alternative. It should be the one which reflects in some way the degree of convergence but is not so rigid as to base everything on the convergence of the population itself. So we adopt the following approach: In each generation, we pick the sample of small size and compute its average performance. If the performance of current structure is close to this average, we increment a counter by 1 (The closeness is decided, for example, by whether they are within a few percent of standard deviation of the initial population). As the counter increases, it shows that the population is converging. If the counter is eventually goes beyond some fraction (in our case, fraction is 10 percent) of generations taken so far, the processor signals that it reached a converged state. If all processors reach this state, the DGA stops.
The advantage of this routine is as follows:

1) It works in noisy environments: It does not matter if the variance fluctuates or not. As long as the instance of close convergence far outnumbers the other, the DGA will stop.

2) It does not stop due to false convergence: When theDGA enters a false convergence state, it does not stop there but goes until the instances of converged states in the processors occurs enough times, does enabling the population to diverge again and produce better structures.

3) It may serve as an general stopping routine since it is problem and problem size independent.

Some of Actual Data Structures in Butterfly Implementation

Here, $CurPop$, $NewPop$,$CurF$ and $NewF$ are implemented as arrays of pointers which points to cur_i, new_i, cv_i, nv_i respectively. That is,

$*CurPop[i] = cur_i$
$*NewPop[i] = new_i$
$*CurF[i] = cv_i$
$*NewF[i] = nv_i$.

Each processor has its local copy of these four arrays. This is again for avoiding contention. In the case of CGA, these pointers are updated synchronously after **Operation** is done. Since each structure is updated asynchronously in DGA, pointers to its storage and to its performance are updated at different times. By knowing the location where a pointer is stored instead of that pointer itself, we can update them locally and still other processor can access correct pointers. Pointers itself can change but the location of their storage remains unchanged.