On Designing Genetic Algorithms for Hypercube Machines

R. Baraglia†, M. Bucci, D. Circelli, R. Perego
Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR),
via S. Maria 36, Pisa, 56126 Italy
† Corresponding and presenting author,
phone: +39 50 593210, fax: +39 50 904052, e-mail r.baraglia@cnuce.cnr.it

Abstract

In this paper we investigate the design of highly parallel Genetic Algorithms. The Traveling Salesman Problem is used as a case study to evaluate and compare different implementations. To fix the various parameters of Genetic Algorithms to the case study considered, the Holland sequential Genetic Algorithm, which adopts different population replacement methods and crossover operators, has been implemented and tested. Both $fine-grained$ and $coarse-grained$ parallel GAs which adopt the selected genetic operators have been designed and implemented on a 128-node nCUBE 2 multicomputer. The $fine-grained$ algorithm uses an innovative *mapping* strategy that makes the number of solutions managed independent of the number of processing nodes used. Complete performance results showing the behavior of Parallel Genetic Algorithms for different population sizes, number of processors used, migration strategies are reported.

1 Introduction

Genetic Algorithms (GAs) [11, 12] are stochastic optimization heuristics in which searches in solution space are carried out by imitating the population genetics stated in Darwin's theory of evolution. Selection, crossover and mutation operators, directly derived by from natural evolution mechanisms are applied to a population of solutions, thus favoring the birth and survival of the best solutions. GAs have been successfully applied to many NP hard combinatorial optimization problems [6], in several application fields such as business, engineering, and science.

In order to apply GAs to a problem, a genetic representation of each individual (*chromosome*) that constitutes a solution of the problem has to be found. Then, we need to create an initial population, to define a cost function to measure the $fitness$ of each solution, and to design the genetic operators that will allow us to produce a new population of solutions from a previous one. By iteratively applying the genetic operators to the current population, the fitness of the best individuals in the population converges to local optima.

Figure 1 reports the pseudo–code of the Holland genetic algorithm. After randomly generating the initial population $\beta(0)$, the algorithm at each iteration of the outer `repeat--until` loop generates a new population $\beta(t+1)$ from $\beta(t)$ by selecting the best individuals of $\beta(t)$ (function `SELECT()`) and probabilistically applying the *crossover* and *mutation* genetic operators. The selection mechanism must ensure that the greater the fitness of an individual A_k is, the higher the probability of A_k being selected for reproduction. Once A_k has been selected, P_C is its probability of generating a son by applying the crossover operator to A_k and another individual A_i, while P_M and P_I are the probabilities of applying respectively, mutation and inversion operators to the generated individual respectively.

The crossover operator randomly selects parts of the parents' *chromosomes* and combines them to breed a new individual. The mutation operator randomly changes the value of a gene (a single bit if the binary representation scheme is used) within the chromosome of the individual to which it is applied. It is used to change the current solutions with in order to avoid the convergence of the solutions to "bad" local optima.

The new individual is then inserted into population $\beta(t+1)$. Two main replacement methods can be used for this purpose. By adopting the *discrete* population model, the whole population $\beta(t)$

```
Program Holland_Genetic_Algorithm;
begin
t=0;
β (t) = INITIAL_POPULATION() ;
repeat
      for i = 1 to number_of_individuals do
            F(A_i) = COMPUTE_FITNESS(A_i);
      Average_fitness = COMPUTE_AVERAGE_FITNESS(F);
      for k = 1 to number_of_individuals do
      begin
            A_k = SELECT(β (t));
            if (P_C > random(0,1)) then
            begin
                  A_i = SELECT(β (t));
                  A_child = CROSSOVER( A_i,A_k );
                  if (P_M > random (0,1)) then MUTATION (A_child);
                  β (t + 1)=UPDATE_POPULATION (A_child);
            end
      end;
t=t+1;
until (end_condition);
end
```

Figure 1: Pseudo–code of the Holland Genetic Algorithm.

is replaced by new generated individuals at the end of the outer loop iteration. A variation on this model was proposed in [13] by using a parameter that controls the percentage of the population replaced at each generation. The *continuous* population model states, on the other hand, that the new individuals are soon inserted into the current population to replace older individuals with worse fitness. This replacement method allows potentially good individuals to be exploited as soon as they become available.

Irrispective of the replacement policy adopted, population $\beta(t+1)$ is expected to contain a greater number of individuals with good fitness than population $\beta(t)$.

The GA end condition can be to reach a maximum number of generated populations, after which the algorithm is forced to stop or the algorithm converges to stable average fitness values.

The following are some important properties of GAs:

- they do not deal directly with problem solutions but with their genetic representation thus making GA implementation independent from the problem in question;
- they do not treat individuals but rather populations, thus increasing the probability of finding good solutions;
- they use probabilistic methods to generate new populations of solutions, thus avoiding being trapped in "bad" local optima.

On the other hand, GAs do not guarantee that global optima will be reached and their effectiveness very much depends on many parameters whose fixing may depend on the problem considered. The size of the population is particularly important. The larger the population is, the greater the possibility of reaching the optimal solution. Increasing the population clearly results in a large increase in GA computational cost which, as we will see later, can be mitigated by exploiting parallelism.

The rest of the paper is organized as follows: Section 2 briefly describes the computational models proposed to design parallel GAs; Section 3 introduces the Traveling Salesman Problem used as our case study, discusses the implementation issues and presents the results achieved on a 128–node hypercube multicomputer; finally Section 4 outlines the conclusions.

2 Parallel Genetic Algorithms

The availability of ever faster parallel computers means that parallel GAs can be exploited to reduce execution times and improve the quality of the solutions reached by increasing the sizes of populations managed.

In [5, 3] the parallelization models adopted to implement GAs are classified. The models described are:

- **centralized model**. A single unstructured *panmitic* population is processed in parallel. A master processor manages the population and the selection strategy and requests a set of slave processors to compute the fitness function and other genetic operators on the chosen individuals. The model scales poorly and explores the solution space like a sequential algorithm which uses the same genetic operators. Several implementations of centralized parallel GAs are described in [1].

- **fine-grained model**. This model operates on a single structured population by exploiting the concepts of *spatiality* and *neighborhood*. The first concept defines that a very small sub-population, ideally just an individual, is stored in one element (node) of the logical connection topology used, while the second specifies that the selection and crossover operators are applied only between individuals located on nearest–neighbor nodes. The neighbors of an individual determine all its possible partners, but since the neighbor sets of partner nodes overlap, this provides a way to spread good solutions across the entire population. Because of its scalable communication pattern, this model is particularly suited for massively parallel implementations. Implementations of fine–grained parallel GAs applied to different application problems can be found in [8, 9, 14, 15, 19].

- **coarse-grained model**. The whole population is partitioned into sub-populations, called *islands*, which evolve in parallel. Each island is assigned to a different processor and the evolution process takes place only among individuals belonging to the same island. This feature means that a greater genetic diversity can be maintained with respect to the exploitation of a panmitic population, thus improving the solution space exploration. Moreover, in order to improve the sub-population genotypes, a migration operator that periodically exchanges the best solutions among different islands is provided. depending on the migration operator chosen we can distinguish between *island* and *stepping stone* implementations. In *island* implementations the migration occurs among every island, while in *stepping stone* implementations the migration occurs only between neighboring islands. Studies have shown that there are two critical factors [10].: the number of solutions migrated each time and the interval time between two consecutive migrations. A large number of migrants leads to the behavior of the island model similar to the behavior of a panmitic model. A few migrants prevent the GA from mixing the genotypes, and thus reduce the possibility to bypass the local optimum value inside the islands. Implementations of coarse grained parallel GAs can be found in [10, 20, 21, 4, 18, 16].

3 Designing parallel GAs

We implemented both $fine-grained$ and $coarse-grained$ parallel GAs applied to the classic Traveling Salesman Problem on a 128–node nCUBE 2 hypercube. Their performance was measured by varying the type and value of some genetic operators. In the following subsection the TSP case study is described and the parallel GA implementations are discussed and evaluated.

3.1 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) may be formally defined as follow: let $C = \{c_1, c_2, ..., c_n\}$ be a set of n cities and $\forall i$, $\forall j$ $d(c_i, c_j)$ the distance between city c_i and c_j with $d(c_i, c_j) = d(c_j, c_i)$. Solving the TSP entails finding a permutation $\pi\prime$ of the cities $(c_{\pi'(1)}, c_{\pi'(2)}, ..., c_{\pi'(n)})$, such that

$$\sum_{i=1}^{n} d(c_{\pi'(i)}, c_{\pi'(i+1)}) \leq \sum_{i=1}^{n} d(c_{\pi^k(i)}, c_{\pi^k(i+1)}) \qquad \forall \pi^k \neq \pi', (n+1) \equiv 1 \qquad (1)$$

According to the TSP *path representation* described in [9], tours are represented by ordered sequences of integer numbers of length n, where sequence $(\pi(1), \pi(2), ..., \pi(n))$ represents a tour joining, in the order, cities $c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(n)}$. The search space for the TSP is therefore the set of all permutations of n cities. The optimal solution is a permutation which yields the minimum cost of the tour.

The TSP instances used in the tests are: **GR48**, a 48-city problem that has an optimal solution equal to 5046, and **LIN105**, a 105-city problem that has a 14379 optimal solution[1].

3.2 Fixing the genetic operators

In order to study the sensitivity of the GAs for the TSP to the settings of the genetic operators, we used Holland's sequential GA by adopting the *discrete generation model*, one and two point crossover operators, and three population replacement criteria.

- The *discrete generation model* separates sons' population from parents' population. Once all the sons' population has been generated, it is merged with the parents' population according to the replacement criteria adopted.
- One point crossover breaks the parent's tours into two parts and recombines them in the son in a way that ensures tour legality [2]. Two points crossover [7] works like the one point version but breaks the parent's tour into three different parts. A mutation operator which simply exchanges the order of two cities of the tour has been also implemented and used [9].
- The replacement criterion specifies a rule for merging current and new populations. We tested three different replacement criteria, called R1, R2 and R3. R1 replaces solutions with lower fitnesses of the current population with all the son solutions unaware of their fitness. R2 orders the sons by fitness, and replaces an individual i of the current population with son j only if the fitness of i is lower than the fitness of j. R2 has a higher control on the population than R1, and allows only the best sons to enter the new population. R3 selects the parents with a lower than average fitness, and replaces them with the sons with above average fitnesses.

The tests for setting the genetic operators were carried out by using a 640 solution population, a 0.2 mutation parameter (to apply a mutation to 20% of the total population), 2000 generations for the 48 city TSP, and 3000 generations for the 105 city TSP. Every test was run 32 times, starting from different random populations, to obtain an average behavior. From the results of the 32 tests we computed:

- the average solution: $AVG = \frac{\sum_{i=1}^{32} F_{E_i}}{32}$, where F_{E_i} is the best $fitness$ obtained with run E_i;
- the best solution: $BST = min\{F_{E_i}, i = 1....32\}$;
- the worst solution: $WST = max\{F_{E_i}, i = 1....32\}$.

These preliminary tests allow us to choose some of the most suitable genetic operators and parameters for the TSP. Figure 2 plots the average fitness obtained by varying the crossover type on the 48–city TSP problem. The crossover was applied to 40% of the population and the R2 replacement criterion was used. As can be seen, the two point crossover converges to better average solutions than the one point operator. The one point crossover initially exhibits a better behavior, but after 2000 generations, converges to solutions that have considerably higher costs. We obtained a similar behavior for the other replacement criteria and for the 105-city TSP.

Table 1 reports AVG, BST and WST results for the 48-city TSP obtained by varying both the population replacement criterion and the percentage of the population to which the two point crossover has been applied. On average, the crossover parameter values in the range $0.4 - 0.6$ lead to better solutions, almost irrispective of the replacement criterion adopted. Figure 3 shows the behavior of the various replacement criteria for a 0.4 crossover value. The R2 and R3 replacement criteria resulted in a faster convergence than R1, and they converged to very near fitnesses.

[1] Both the TSP instances are available at: ftp://elib.zib-berlin.de/pub/mp-testdata/tsp/tsplib.html

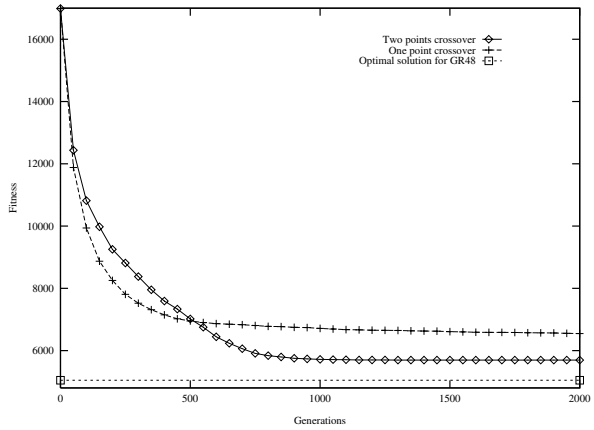

Figure 2: Fitness values obtained with the execution of the sequential GA on the 48–city TSP by varying the crossover operator, and by using the R2 replacement criteria.

		Crossover parameter			
		0.2	0.4	0.6	0.8
R1	**AVG**	6255	5632	5585	5870
	BST	5510	5315	5135	5305
	WST	7828	6079	6231	6693
R2	**AVG**	5902	5696	5735	5743
	BST	5405	5243	5323	5410
	WST	7122	6180	6225	6243
R3	**AVG**	6251	5669	5722	5773
	BST	5441	5178	5281	5200
	WST	7354	6140	6370	6594

Table 1: Fitness values obtained with the execution of the sequential GA on the 48–city TSP by varying the value of the crossover parameter and the population replacement criterion.

3.3 The coarse grained implementation

The coarse grained parallel GA was designed according to the *discrete generation* and *stepping stone* models. Therefore, the new solutions are merged with the current population at the end of each generation phase, and the migration of the best individuals among sub-population is performed among ring-connected islands. Each of the P processors manages N/P individuals, with N population size (640 individuals in our case). The number of migrants is a fixed percentage of the sub-population. As in [4], migration occurs periodically in a regular time rhythm, after a fixed number of generations.

In order to include all the migrants in the current sub-populations, and to merge the sub-population with the locally generated solutions, R1 and R2 replacement criteria were used, respectively. Moreover, the two point crossover operator was adopted.

Table 2 reports some results obtained by running the coarse grained parallel GA on the 48–city TSP. **M** denotes the migration parameter. The same data for a migration parameter equal to 0.1 are plotted in Figure 4. It can be seen that AVG, BST and WST solutions get worse values by increasing the number of the nodes used. This depends on of the constant population size used: with 4 nodes sub-populations of 16 solutions are exploited, while with 64 nodes the sub-populations only consists of 10 individuals. Decreasing the number of solutions that forms a sub-population worsens the search in the solution space; small sub-populations result in an insufficient exploration of the solution space. The influence of the number of migrants on the convergence is clear from Table 2. When the sub-populations are small, a higher value of the migration parameter may improve the quality of solutions through a better mix of the genetic material.

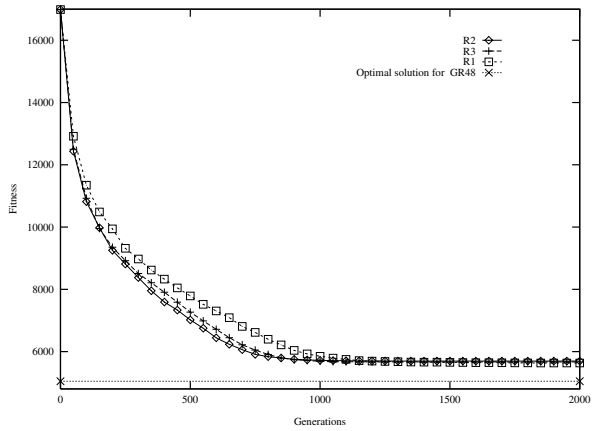

Figure 3: Fitness values obtained by executing the sequential GA on the 48–city TSP with a 0.4 crossover parameter, and by varying the replacement criterion.

		Number of processing nodes					
		4	*8*	*16*	*32*	*64*	*128*
M=0.1	**AVG**	5780	5786	5933	6080	6383	6995
	BST	5438	5315	5521	5633	5880	6625
	WST	6250	6387	6516	6648	8177	8175
M=0.3	**AVG**	5807	5877	5969	6039	6383	6623
	BST	5194	5258	5467	5470	5727	6198
	WST	6288	6644	7030	6540	8250	7915
M=0.5	**AVG**	5900	5866	5870	6067	6329	6617
	BST	5419	5475	5483	5372	6017	6108
	WST	6335	6550	7029	6540	8250	7615

Table 2: Fitness values obtained with the execution of the coarse grained GA on the 48-city TSP by varying the migration parameter.

3.4 The fine grained implementation

The fine grained parallel GA was designed according to the *continuous generation model*, which is much more suited for fine grained parallel GAs than the *discrete* one. The two point crossover operator was applied.

According to the fine grained model the population is structured in a logic topology which fixes the rules of interaction between the solution and other solutions: each solution s is placed at a vertex $v(s)$ of logic topology T. The crossover operation can only be applied among nearest neighbor solutions placed on the vertices directly connected in T. Our implementation exploits the physical topology of the target multicomputer, therefore the population of 2^N individuals is structured as a N-dimensional hypercube.

By exploiting the recursivity of the hypercube topology definition, we made the number of solutions treated independent of the number of nodes used to execute the algorithm. As can be seen in Figure 5, a $2^3 = 8$ solution population can be placed on a $2^2 = 4$ node hypercube, using a simple mapping function which masks the first (or the last) *bit* of the **Grey** code used to numerate the logical hypercube vertices [17]. Physical node $X00$ will hold solutions 000 and 100 of the logic topology, not violating the neighborhood relationships fixed by the population structure. In fact, the solutions on the neighborhood of each solution s will still be in the physical topology on directly connected nodes or on the node holding s itself.

We can generalize this *mapping* scheme: to determine the allocation of a 2^N solution population on a 2^M node physical hypercube, with $M < N$, we simply mask the first (the last) $N - M$ *bits* of

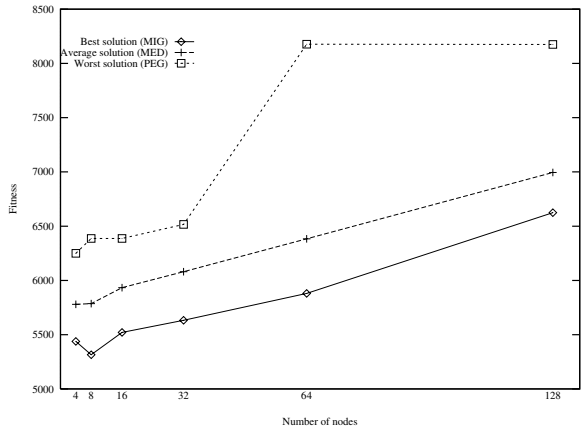

Figure 4: AVG, BST and WST values obtained by executing the coarse grained GA on the 48–city TSP as a function of the number of nodes used, and 0.1 as migration parameter.

the binary coding of each solution.

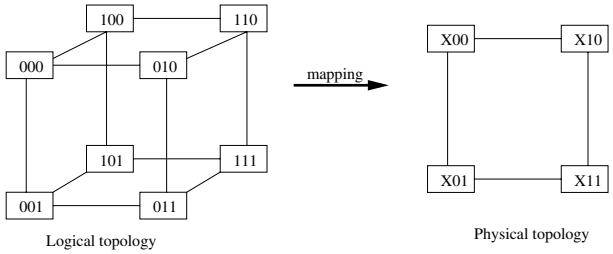

Figure 5: Example of an application of the *mapping* scheme.

Table 3 reports the fitness values obtained with the execution of the fine grained GA by varying the population dimension from 128 solutions (a 7 dimension hypercube) to 1024 solutions (a 10 dimension hypercube). As expected, the ability to exploit population sizes larger than the number of processors used in our mapping scheme, leads to better quality solutions especially when few processing nodes are used. The improvement in the fitness values by increasing the number of nodes while maintaining the population size fixed, is due to a particular feature of the implementation, which aims to minimize the communication times to the detriment of the diversity of the same node solutions. Selection rules tend to choose partner solutions in the same node. The consequence is a greater uniformity in solutions obtained on few nodes, which worsens the exploration of the solution space. The coarse grained implementation suffered of the opposite problem which resulted in worse solutions obtained as the number of nodes was increased. This behavior can be observed by comparing Figure 4, concerning the coarse grained GA, and Figure 6, concerning the fine grained algorithm with a 128 solution population applied to the 48–city TSP.
Table 4 shows that an increase in the number of solutions processed results in a corresponding increase in the speedup values. This is because a larger number of individuals assigned to the same processor leads to lower communication overheads for managing the interaction of each individual with neighbor partners.

3.5 Comparisons

We compared the fine and coarse grained algorithms on the basis of the execution time required and the fitness values obtained by each one after the evaluation of 512000 solutions. This comparison criterion was chosen because it allows to overcome computational models diversity that make non

Population size		Number of processing nodes						
		1	*4*	*8*	*16*	*32*	*64*	*128*
128	**AVG**	39894	24361	23271	23570	23963	22519	22567
	BST	34207	20774	19830	20532	21230	20269	20593
	WST	42127	30312	26677	27610	27634	28256	25931
256	**AVG**	33375	25313	22146	21616	21695	22247	21187
	BST	29002	24059	20710	19833	20144	19660	19759
	WST	40989	26998	23980	24007	23973	24337	22256
512	**AVG**	28422	23193	22032	21553	20677	20111	20364
	BST	28987	22126	19336	20333	19093	18985	18917
	WST	41020	25684	23450	22807	22213	21696	21647
1024	**AVG**	25932	23659	22256	20366	19370	18948	19152
	BST	27010	21581	21480	18830	18256	18252	17446
	WST	40901	25307	22757	21714	20766	19525	20661

Table 3: Fitness values obtained with the execution of the fine grained GA applied to the 105-city TSP after 3000 generations, by varying the number of solutions per node.

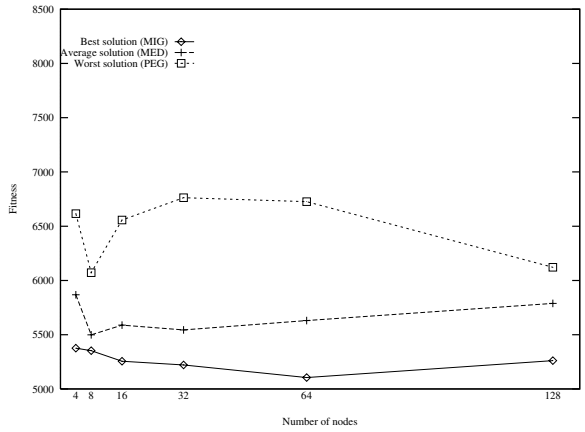

Figure 6: AVG, BST and WST values obtained by executing the fine grained GA applied to the 48-city TSP as a function of the number of nodes used. The population size was fixed to 128 individuals.

comparable the fine and coarse grained algorithms. The evaluation of 512000 new solutions allows both the algorithms to converge and requires comparable execution times. Table 5 shows the results of this comparison. It can be seen that when the number of the nodes used increases the fine grained algorithm gets sensibly better results than the coarse grained one. On the other hand, the coarse grained algorithm shows a super–linear speedup due to the *quick sort* algorithm used by each node for ordering by fitness the solutions managed. As the number of nodes is increased, the number of individuals assigned to each node decreases, thus requiring considerably less time to sort the sub-population.

4 Conclusions

We have discussed the results of the application of parallel GA algorithms to the TSP. In order to analyze the behavior of different replacement criteria and crossover operators and values Holland's sequential GA was implemented. The tests showed that the two point crossover finds better solutions, as does a replacement criteria which replaces an individual i of the current population with son j only if the fitness of i is worse than the fitness of j. To implement the $fine-grained$ and $coarse-grained$ parallel GAs on a hypercube parallel computer the most suitable operators were adopted. For the

Number of nodes	Speedup			
	128 individuals	256 individuals	512 individuals	1024 individuals
1	1	1	1	1
4	3.79	3.84	3.89	3.92
8	7.25	7.47	7.61	7.74
16	13.7	14.31	14.78	15.12
32	25.25	27.02	28.03	29.38
64	44.84	49.57	53.29	56.13
128	78.47	88.5	98.06	105.32

Table 4: Speedup of the of the fine grained GA applied on the 105-city TSP for different population sizes.

	Number of processing nodes					
	4	*8*	*16*	*32*	*64*	*128*
Fine grained AG						
AVG	26373	26349	25922	25992	24613	23227
BST	23979	23906	23173	21992	22145	20669
WST	30140	27851	29237	29193	30176	2680
Execution times	1160	606	321	174	98	51
Coarse grained AG						
AVG	23860	24526	27063	29422	32542	35342
BST	20219	21120	23510	25783	30927	33015
WST	25299	29348	36795	39330	39131	41508
Execution times	1670	804	392	196	97	47

Table 5: Fitness values and execution times (in seconds) obtained by executing the fine and coarse grained GA applied to the 105-city TSP with a population of 128 and 640 individuals, respectively.

coarse grained GA we observed that the quality of solutions gets worse if he number of nodes used is increased. Moreover, due to the sorting algorithm used to order each sub-population by fitness, the speedup of the coarse grained GA were super–linear. Our fine-grained algorithm adopts a mapping strategy that allows the number of solutions to be independent of the number of nodes used. The ability to exploit population sizes larger than the number of processors used gives better qualitysolutions especially when only a few processing nodes are used. Moreover, the quality of solutions does not get worse if the number of the nodes used is increased. The fine grained algorithm showed good scalability. A comparison between the fine and coarse grained algorithms highlighted that fine grained algorithms represent the better compromise between quality of the solution reached and the execution time spent on finding it.
The AGs implemented reached only "good" solutions. In order to improve the quality of solutions obtained, we are working to include a local search procedure within the AG.

References

[1] R. Bianchini and C.M. Brown. Parallel genetic algorithm on distribuited-memory architectures. Technical Report TR 436, Computer Sciences Department University of Rochester, 1993.

[2] H. Braun. On solving travelling salesman problems by genetic algorithms. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN , volume 496 of Lecture Notes in Computer Science*, pages 129–133. Springer-Verlag, 1991.

[3] E. Cantu-Paz. A summary of research on parallel genetic algoritms. Technical Report 95007, University of Illinois at Urbana-Champaign, Genetic Algoritms Lab. (IlliGAL), http://gal4.ge.uiuc.edu/illigal.home.html, July 1995.

[4] S. Cohoon, J. Hedge, S. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. *IEEE Transaction on CAD*, 10(4):483–491, April 1991.

[5] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In *Parallel Genetic Algorithms*, pages 5–42. IOS Press, 1993.

[6] M. R. Garey and D.S. Jonshon. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[7] D. Goldberg and R. Lingle. Alleles, loci, and the tsp. In *Proc. of the First International Conference on Genetic Algorithms*, pages 154–159, 1985.

[8] M. Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN*, volume 496, pages 398–406. Lecture Notes in Computer Science, 1990.

[9] M. Gorges-Schleuter. *Genetic Algoritms and Population Structure*. PhD thesis, University of Dortmund, 1991.

[10] P. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. PhD thesis, University of Michigan, 1985.

[11] J. Holland. *Adaptation in Natural and Artificial Systems*. Univ. of Mitchigan Press, 1975.

[12] J.H. Holland. Algoritmi genetici. *Le Scienze*, 289:50–57, 1992.

[13] K.A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975.

[14] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433. Morgan Kaufmann Publishers, 1989.

[15] H. Muhlenbein. Parallel genetic algorithms, population genetic and combinatorial optimization. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN*, volume 496, pages 407–417. Lecture Notes in Computer Science, 1991.

[16] H. Muhlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.

[17] nCUBE Corporation. ncube2 processor manual. 1990.

[18] C. Pettey, M. Lenze, and J. Grefenstette. A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 155–161. L. Erlbaum Associates, 1987.

[19] M. Schwehm. Implementation of genetic algorithms on various interconnections networks. *Parallel Computing and Transputer applications*, pages 195–203, 1992.

[20] R. Tanese. Parallel genetic algorithms for a hypercube. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 177–183. L. Erlbaum Associates, 1987.

[21] R. Tanese. Distribuited genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–440. M. Kaufmann, 1989.