Journal of Parallel and Distributed Computing **62**, 1362–1385 (2002)
doi:10.1006/jpdc.2002.1851

Heterogeneous Computing and Parallel Genetic Algorithms

Enrique Alba,[1] Antonio J. Nebro, and José M. Troya

Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, E.T.S.I. en Informática, Campus de Teatinos, Málaga 29071, Spain
E-mail: eat@lcc.uma.es; antonio@lcc.uma.es; troya@lcc.uma.es

Received March 29, 2000; accepted January 29, 2002

This paper analyzes some technical and practical issues concerning the heterogeneous execution of parallel genetic algorithms (PGAs). In order to cope with a plethora of different operating systems, security restrictions, and other problems associated to multi-platform execution, we use Java to implement a distributed PGA model. The distributed PGA runs at the same time on different machines linked by different kinds of communication networks. This algorithm benefits from the computational resources offered by modern LANs and by Internet, therefore allowing researchers to solve more difficult problems by using a large set of available machines. We analyze the way in which such heterogeneous systems affect the genetic search for two problems. Our conclusion is that super-linear performance can be achieved not only in homogeneous but also in heterogeneous clusters of machines. In addition, we study some special features of the running platforms for PGAs, and basically find out that heterogeneous computing can be as efficient or even more efficient than homogeneous computing for parallel heuristics. © 2002 Elsevier Science (USA)

Key Words: parallel genetic algorithms; Java; heterogeneous computational systems; speedup.

1. INTRODUCTION

Evolutionary algorithms (EAs) are modern techniques for searching complex spaces for an optimum [5]. These meta-heuristics can be differentiated into a quite large set of algorithmic families representing bio-inspired systems which mimic natural evolution. They are said to be "evolutionary" because their basic behavior is drawn from Nature, and evolution is the basic natural force driving a population of tentative solutions towards the problem regions where the optimal solutions are located.

[1] To whom correspondence should be addressed.

Parallel EAs (PEAs) are becoming an important branch of research since they provide a faster and more efficient way of solving known and new problems. There exist many useful models of EAs; here, we mainly focus on a kind of EA known as *genetic algorithm* (GA) [12, 17, 22]. This paper deals with some novel aspects of parallel genetic algorithms (PGAs) relating their execution in heterogeneous clusters of machines.

As an EA, a GA is a randomized optimization method that uses information on the problem to guide the search (see Algorithm 1). At each generation (iteration) t, they operate on a population of individuals $P(t)$, each one encoding a sub-optimal solution, thus performing a search from many zones of the problem space at the same time; this is a salient feature of EAs not found in more classical methods such as simulated annealing or other gradient-descent techniques. Each individual is a string of symbols encoding a solution for the problem plus an associated fitness value. This fitness value is computed by the objective function, and it is intended to rank the quality of the evaluated individual with respect to the rest of the population. The application of simple stochastic variation operators (mixing parts of two strings, randomly changing their contents, etc.) leads this population towards fittest regions in an iterative manner, until a stopping criterion is fulfilled (e.g., an optimum is found).

Algorithm 1 (*Sequential Genetic Algorithm*)

```
t := 0;
initialize & evaluate[$P(t)$] ;
**while not** stop_condition **do**
    $P'(t)$ := variation [$P(t)$] ;
    evaluate [$P'(t)$] ;
    $P(t+1)$ := select[$P'(t)$] ;
    t := t + 1 ;
**end while**
```

Binary, integer, and also real strings are commonly used in GAs. Every string must have its own fitness value indicating how good it is. A GA searches for progressively fitter strings by following the loop of Algorithm 1, thus creating populations of increasing mean fitness.

It is very usual to find parallel implementations of GAs in the literature; for example, see an up-to-date survey of PGAs in [2]. Since genetic algorithms deal in general with a set of sub-optimal solutions known as the population, it is common sense to think in splitting the population into parallel sub-algorithms as a way to improve efficiency; there are many different milestone works confirming this feeling, such as [7, 21, 23, 26, 27, 30]. But making a GA *faster* is not the only advantage that can be expected when designing a parallel GA. A PGA has an improved *power* to tackle more complex problems since it can use more memory and CPU resources. The *versatility* is larger than in a sequential version since different solution species can evolve and coexist in parallel, thus yielding alternative solutions to the same problem. An additional advantage is that the string *diversity* is better sustained when run in parallel, which is good for tackling *new domains* such as non-stationary problems where the optimum varies and moves during the search. Finally, parallel GAs reduce the probability of getting stuck in a local optimum (*efficacy*) and they provide a clear way to interact in parallel with other optimization algorithms (enhanced *applicability*).

Our motivation in this paper is to analyze the behavior of parallel GAs when executed in a heterogeneous software and hardware environment. This kind of systems is usually found in modern labs, where many different types of machines coexist. We want to find out whether existing heterogeneous computing resources can be also used to efficiently solve optimization problems with PGAs. Interesting changes in the run time and in the numerical results are expected, since different machines are running different operating systems on different processors with different memory capacities. Therefore, the results can differ from the rest of (better known) homogeneous studies with PGAs.

We use several clusters of similar machines as well as isolated machines. Since the possible combinations of hardware and software are quite large, we will encompass a structured presentation of our results. First, we analyze uniprocessor execution, then we study multiprocessor homogeneous execution, and finally we show some particular results over heterogeneous executions by merging all the processors in a single pool of computational resources used to solve our problems with PGAs.

We begin by measuring basic parameters such as the run time, effort to locate a solution, speedup, number of evaluations per second, and some other related values. Rather than using complex problems and efficiently computing perfect numerical solutions, our goal is to offer a thorough study on executing PGAs in heterogeneous clusters of computers.

Since we deal with very different hardware and software we have chosen Java as the implementation language. Our reasons are that, first, Java is inherently platform-independent; second, the communication libraries are a part of the standard language definition (thereby making unnecessary to include foreign communication libraries); and, finally, Java allows an easy interaction among machines having different data representations. Also, many advances and Internet issues are daily crafted by using Java, thus improving compilers' efficiency and importance.

This paper is organized as follows. The next section includes a quick review of decentralized and parallel GAs to offer a brief guide to non-specialists. In Section 3 we discuss some classical performance measures and their extension to the PGA field. Section 4 explains the test suite and the PGA being used for the experiments. Section 5 contains the analysis of the tests. Finally, some concluding remarks and future research lines are presented in Section 6.

2. AN INTRODUCTION TO DECENTRALIZED AND PARALLEL GAS

The existing separate studies with PGAs pose some common problems to researchers. The first problem comes from the high variety of different parallel implementations. Some of these implementations have the same behavior as a sequential GA, although others do not. It is only in these last years that an effort is being made to propose unified methodologies for designing and studying sequential and parallel GAs [2, 28].

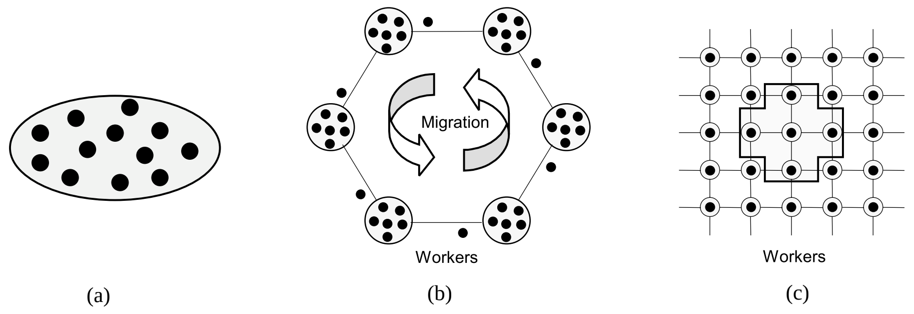

FIG. 1. A panmictic GA (a), and two structured GAs: distributed (b) and cellular (c).

A textbook GA uses a single population to which it applies variation and selection operators (see Algorithm 1 again) [5]. These algorithms are thus called *panmictic* GAs (Fig. 1a). Panmictic GAs can be readily parallelized by using a master/slave model in order to perform evaluations in parallel, but retaining its panmictic behavior. This is called a *global parallelization* of the algorithm [2] (not used here but useful in many CPU intensive tasks). This PGA works well for a small number of nodes, but becomes hamstrung by excessive communications as the number of nodes increases.

In short, a *panmictic* GA has all its strings (black points in Fig. 1a) in a single population, and, consequently, each of them can potentially mate with any other. Structuring the population usually leads to distinguish between *distributed* (Fig. 1b) and *cellular* (Fig. 1c) GAs. These two last models are usually found implemented on MIMD and SIMD machines, respectively. However, nothing prevents other kinds of parallelization or even a sequential time-sliced simulation to be used. Therefore, we distinguish between two kinds of structured GAs: *distributed* (dGAs) and *cellular* (cGAs) GAs, which could in turn be parallelized (or not) as coarse and fine grain PGAs, respectively.

In Fig. 2 we show what we call the structured-population evolutionary algorithm cube. This cube provides a generalized way to classify structured EAs and thus GAs. While a distributed GA has a large sub-population ($\gg 1$), a cGA has typically one single string in every sub-algorithm. In a dGA, the sub-algorithms are loosely

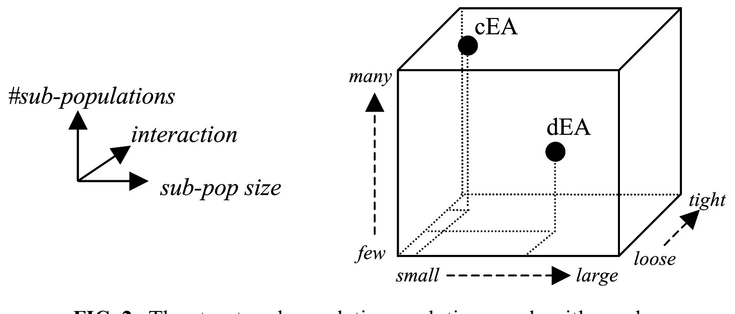

FIG. 2. The structured-population evolutionary algorithm cube.

connected, while they are tightly coupled in a cGA. Additionally, in a dGA there exist only a few sub-algorithms, while in a cGA there is a large number of them.

For parallel GAs, a unified approach is directly tackled rarely [3, 28]. However, this is very important since we can study the full spectrum of parallel implementations by only considering different coding, operators, and communication details with respect to a unified model. In fact, this is a natural vision of PGAs that is inspired in the initial work of Holland [17], in which a *grained adaptive system* is a set of algorithmic grains with shared structures. Also, Gordon and Whitley [13] made concrete contributions relating the ways in which parallelization affects these and other GAs.

Since GAs are very popular and because we are going to use a PGA in this paper, let us see the outline of a general PGA in Algorithm 2. It begins by randomly creating a population $P(t=0)$ of μ structures (strings). Each string encodes (genotype) the p problem variables, usually as a vector over $B=\{0,1\}$ or R, therefore yielding $I=B^{p\,l_x}$ and $I=R^p$, respectively, where l_x is the number of bits used to encode each variable in the string. An evaluation function Φ is needed to associate a quality real value to every structure, called its *fitness* value. The stopping criterion ι of the reproductive loop is to fulfill some condition: max-iterations, solution found, etc. The solution is identified as the best structure ever found.

Algorithm 2 (*Parallel Genetic Algorithm*)

```
SUBALGORITHM #i
t := 0;
initialize: $P(0) = \vec{a}_1(0), \ldots, \vec{a}_\mu(0) \in I^\mu$;
evaluate: $P(0) : \Phi(\vec{a}_1(0)), \ldots, \Phi(\vec{a}_\mu(0))$;
while not $\iota(P(t))$ do        // Reproductive Loop
    select:      $P'(t) := s_{\Theta_s} P(t)$ ;
    recombine:   $P''(t) := \oplus_{\Theta_c} P('t)$ ;
    mutate:      $P'''(t) := m_{\Theta_m} P(''t)$ ;
    evaluate:    $P'''(t) : \Phi(\vec{a}_1'''(t)), \ldots, \Phi(\vec{a}_\lambda'''))$ ;
    replace:     $P(t+1) := r_{\Theta_r}(P'''(t) \cup Q)$ ;
    ⟨communication with neighbors (sync/async)⟩
    t := t + 1 ;
end while
```

The selection s_{Θ_s} operator makes copies of every structure from $P(t)$ to $P'(t)$ subject to the fitness values of the structures (some parameters Θ_s might be required for controlling this operation). The typical variation operators in GAs are *crossover* ($\oplus$), which exchanges string slices, and *mutation* (m), which randomly alters symbols in the strings. They both are stochastic techniques whose behavior is governed by a set of parameters e.g., an application probability: $\Theta_c = \{p_c\}$ usually high for crossover, and $\Theta_m = \{p_m\}$ usually small for mutation. Crossover is a binary operator, while mutation is a unary one.

Finally, each iteration ends by selecting μ (out of λ) individuals for the new population (*replacement policy r*). For this purpose, the new pool $P'''(t)$ plus a set Q are considered. The best structure usually survives deterministically (*elitism*). Many panmictic variants exist, but two of them are especially popular [29]: the *generational*

GA —genGA— ($\lambda = \mu$, $Q = \emptyset$), and the *steady-state GA* —ssGA— ($\lambda = 1$, $Q = P(t)$). In a generational GA a full new population replaces the old one, while in a steady-state GA one or two individuals are created and immediately inserted within the population, thus merging parents and children in the same population.

In a parallel GA there exist many elementary sub-algorithms (*island* GAs or *demes*) working on separate populations $P(t)$. All these sub-algorithms are intended to perform the same reproductive plan, otherwise the PGA is *heterogeneous*. We focus on homogeneous PGAs, where each island performs the same computations than the rest. This may not be confused with the hardware being used to run the algorithm (which in turn can be homo/heterogeneous).

Each sub-algorithm includes an additional phase of periodic *communication* with a set of neighboring sub-algorithms located on some topology. This communication usually consists in exchanging a set of individuals or statistics. This can be made in a synchronous manner, thus making parallel migrations at the same time in each island, or asynchronously, and then the incoming individual is incorporated whenever it arrives. The characteristics of this communication step (frequency, neighborhood, policies for sharing individuals, etc.) define whether it is a distributed, a cellular, or a hybrid GA. Many works have found interesting advantages in using the asynchronous execution model [3, 24]; also, since we are going to use heterogeneous hardware, it seems appropriate to use asynchronous PGAs to avoid bottlenecks caused by the slowest processors.

3. PERFORMANCE MEASURES FOR PGAS

In this section, our goal is to present the parameters we are going to use for assessing the performance of a PGA. Section 3.1 is devoted to discuss general parameters for evaluating numeric and real-time efficiency, while Section 3.2 gets deeper on the understanding of speedup in the evolutionary computation field.

3.1. Numeric and Real-Time Measures

A well-accepted way of measuring the numerical performance of an EA is to check the number of evaluations of the objective function needed to locate the optimum. We call this the *numeric effort* of the algorithm. Another interesting measure for PEAs is the total run time needed to locate a solution by the distributed algorithm. In this last case, we must use average times of many independent runs (30 runs provide good estimates), since we are dealing with a non-deterministic algorithm.

Many authors hold that measuring run time is the "ultimate way of comparing and studying such algorithms" [16]; nevertheless, we also will study their speedup [1]. Speedup must be measured carefully in the PGA field. First, we must use average times and not absolute ones, due to the mentioned non-deterministic behavior of a PGA. Second, the uni and multiprocessor algorithms must be exactly the same, thus avoiding the use of a panmictic GA in the uniprocessor case plus a distributed one in the multiprocessor case. Third, for a meaningful speedup, we must run the PGA until a solution for the problem is found, as it seems common sense (and as some authors have extremely enforced in the past [3, 8, 16]). After fulfilling all these requirements, speedup will provide us with a meaningful and fair value indicating the advantage of using more processors for running the PGA.

For the discussion, we shall assume that we are using in parallel M machines with identical processors, and that the execution time for an algorithm using $m \leqslant M$ processors is T_m. The PGA-adapted definition of speedup computes the ratio between the mean execution time on a uniprocessor $E[T_1]$ and the mean execution time on m processors $E[T_m]$:

$$s_m = \frac{E[T_1]}{E[T_m]}. \tag{1}$$

With this definition we can distinguish amongst:

$$\begin{array}{lcl} \text{Sub-linear speedup} & \Leftrightarrow & s_m < m, \\ \text{Linear speedup} & \Leftrightarrow & s_m = m, \\ \text{Super-linear speedup} & \Leftrightarrow & s_m > m. \end{array} \tag{2}$$

The aim of a parallel program is to reduce the time to find a solution to a given problem. Ideally, we would expect m processors to generate the solution m times as quickly as a single processor. This is termed *linear speedup*. For most practical algorithms, we expect T_m to be greater than T_1/m. Only algorithms which may be described as *embarrassingly parallel* can be expected to produce speedup which is perfectly linear with the number of processors. *Near-linear* speedup, a subclass of sub-linear speedup, where $E[T_m]$ is within say 90% of $E[T_1/m]$, is however often possible.

An open research line is the study of speedup in the heterogeneous cluster case. It is likely that performance figures should use wall clock time rather than CPU time, since there is an obvious problem with defining a reference point against which the results should be compared [14]. Our reference point is the execution time of the program on the fastest single processor [9]. Using this measure, we can at least assume that a speedup greater than unity implies that the parallel system is faster than the uniprocessor one. In this same work, Donalson et al. have shown that there is no theoretical upper limit on heterogeneous speedup, which can explain a super-linear result.

Finally, Karp and Flatt [19] have devised another metric for measuring the performance of a general parallel algorithm that can help us to identify much more subtle effects than by using speedup alone. They call it the *serial fraction* of the algorithm (f_m), whose value should remain constant in an ideal system:

$$f_m = \frac{1/s_m - 1/m}{1 - 1/m}. \tag{3}$$

If a speedup value is small (or efficiency is relatively far from 100%, say 87%) we can still say that the result is good if f_m remains constant for different values of m, since the loss of efficiency is due to the limited parallelism of the program. On the other side, smoothly increasing f_m is a warning that the granularity of the parallel tasks is too fine. A third scenario is possible in which a significant reduction in f_m occurs, indicating something akin to super-linear speedup. Karp and Flatt have reported this scenario with Convex C-200 family processors running the Linpack benchmark report. The winners of the Gordon Bell Award [15] also reported this last reduction for three problems: wave motion, fluid dynamics, and beam stress, with up to 1024 processors in a hypercube.

In this paper, we will analyze PGA performances by measuring the speedup and also the serial fraction, in order to enrich our understanding of the effects of parallelism on this family of heuristics. Other values, such as the number of steps per second performed by the algorithm, are used here to shed some light in comparing results on different hardware configurations.

3.2. Is It Really Possible to Have Super-Linear Speedup in PEAs?

A number of authors have analyzed PGAs attending to different criteria, and many of them came out with a super-linear speedup result when using a parallel machine [4, 6]. On the other hand, based on our past experience with PGAs [2, 3], we expect to get super-linear speedup results sometimes (both in the homogeneous and the heterogeneous cases). Consequently, we discuss here an intriguing point: which are the sources of super-linear speedup in PGAs?

In general, we classify them into *implementation*, *numerical*, and *physical* sources:

- *Implementation source*: The algorithm being run on one processor is inefficient in some way. For example, the complexity of the operators (selection, crossover) reduces dramatically when the population is split and the resulting chunks are dealt with in parallel. If the algorithm run sequentially is panmictic, then this is the reason why the parallel one most probably will throw super-linear performance. On the other hand, if the sequential algorithm uses linear lists of data, the parallel one is faster because it manages shorter lists and merges the results. Super-linear speedup would not occur in this last case if the sequential algorithm would use e.g., a binary data tree [14]. In the case of heterogeneous computers, using different compilers (or different versions/implementations e.g., of a Java interpreter) can also be a source for super-linear speedup.
- *Numerical source*: Since the search space is usually very large, the sequential program may have to search a large proportion of it in order to find the required solution, while the parallel version may, however, find the solution more quickly due to the change in the order in which the space is searched. This holds for EAs as well as for other algorithms such as branch-and-bound [20]. In fact, heuristics, and thus EAs, have a non-zero probability of finding a solution after T s, for any $T > 0$. Providing more processors enlarges this probability of reducing the time to find a solution in the PEA with respect to the same algorithm on one processor. This has been proven by Shonkwiler [26], where we can even find a numerical model for describing the speedup in PEAs with the expression:

$$s_m = m\, a^{m-1} \tag{4}$$

  in which he introduced an acceleration factor a that can explain a super-linear speedup ($a > 1$).

- *Physical source*: When moving from a sequential machine to a parallel one, it is often not only the CPU power which is increased. Frequently, other resources such as memory, caché, etc. increase linearly with the number of processors. The parallel algorithm may be able to achieve super-linear speedup by taking advantage of these additional resources. In heterogeneous clusters the resources are different from each other, with unseen results.

We then conclude that super-linear speedup is possible not only from a theoretical point of view but also as a result of empirical tests, both in homogeneous [26] and heterogeneous [9] parallel computing. As a matter of fact, it is usual to find super-linear performances in meta-heuristics, and especially in parallel GAs; this section has been devoted to clarify such points for readers not familiar to these facts.

4. BENCHMARK, PGA, AND HETEROGENEOUS COMPUTING RESOURCES

In this section, we include both the addressed optimization tasks and the kind of PEA and computers we are using. Our aim is to locate in the same part of the paper the information to make the experiments understandable and susceptible of being reproduced in the future.

4.1. Benchmark

Our analysis will use two problems, namely ONEMAX and P-PEAKS. They both are sought for a *maximum*. The ONEMAX problem has been very useful in the whole EA literature since its simplicity has allowed to derive canonical results and closed forms for many mathematical models (e.g. [25]). The P-PEAKS function [18] represents quite well the new wave of problem generators, and it will allow us to define a difficult and time-consuming problem without introducing any bias or help into our canonical PGA. We now give some more details.

The first function is used only to provide a basic simple case for drawing conclusions on the homogeneous and heterogeneous results. The ONEMAX function is easy (no parameter correlation—epistasis) and continuous. It consists in maximizing the following expression:

$$f_{\mathrm{ONEMAX}}(\vec{x}) = \sum_{i=1}^{n} x_i. \tag{5}$$

The second optimization problem is actually a problem generator proposed in [18]. A problem generator is an easily parameterizable task which has a tunable degree of epistasis, consequently allowing us to derive instances with growing difficulty at will. Also, using a problem generator removes the opportunity to hand-tune algorithms to a particular problem since every independent runs solves a different problem instance, therefore allowing a larger fairness when comparing algorithms. With a problem generator we evaluate our algorithms on a high number of random problem instances, thus increasing the predictive power of the results for the problem class as a whole. Here we utilize the multimodal generator we call P-PEAKS [18].

The idea is to generate a set of P random N-bit strings that represent the location of P peaks in the search space. To evaluate an arbitrary bit string, first locate the nearest peak in Hamming space. Then the fitness of the bit string is the number of bits the string has in common with that nearest peak, divided by N (Eq. (6)). Problems with a small/large number of peaks are weakly/strongly epistatic. Also, a large number of peaks induce a very high time-consuming algorithm, since evaluating every string is computationally hard. Our instance uses $P = 512$ peaks of $N = 512$ bits each, which represents a highly epistatic problem and a high computation time. In fact, a simpler instance having $P = 200$ peaks and $N = 100$ bits per string is considered to produce a considerably difficult problem in [18]. Therefore, this allows the complementary point of analysis with respect ONEMAX for which every evaluation can be completed much faster.

$$f_{P\text{-PEAKS}}(\vec{x}) = \frac{1}{N}\max_{i=1}^{p}\ \{N - Hamming\,D(\vec{x}, Peak_i)\}. \tag{6}$$

4.2. Our Parallel Evolutionary Algorithm

The PEA used in this paper is a distributed genetic algorithm which we will call *dssGA*, since every island is performing a steady-state GA [29]. The steady-state island main loop creates one single individual in every step by applying binary tournament selection of two parents, two point crossover, bit-flip mutation, and replacement of the worst one only if the new one is better than it [3]. See and example of the basic step in Fig. 3. We will use eight islands in this study.

The resulting distributed $(\mu + 1)$-EA is somewhat similar to GENITOR II [31]. The islands are located in a unidirectional ring, and every island sends one copy of a random string to its neighbor in the ring with a given migration frequency (every 32 generations i.e., every $32 \times 512/8$ evaluations). The target island incorporates this string only if it is better than its actual worse string. The reception of incoming strings is asynchronous, since many works have demonstrated that asynchronous

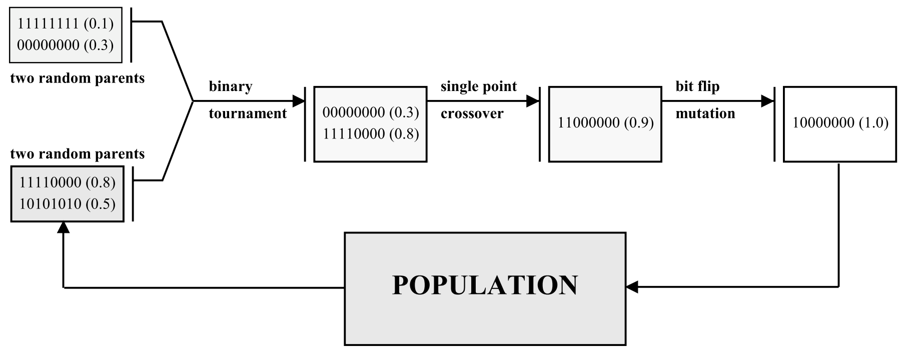

FIG. 3. Numeric computations in a basic step of an island steady-state GA.

PGAs have advantages in reducing the run time [3, 16] and in flexibility [24]. The stopping rule for all the tests is to find a global optimum. All the algorithms are implemented in Java.

We always use a total of 512 individuals, separated into 8 islands (64 individuals per island). It is a homogeneous PGA in which all the islands perform the before mentioned proportional selection, two-point crossover with probability $p_c = 0.85$ and bit-flip mutation with probability $p_m = 1/l$, being l the string length ($l = 512$ for ONEMAX and P-PEAKS).

All the results provided here are the mean of 30 independent runs. Since we include systematic homogeneous and also heterogeneous results with two tasks (the second one very time consuming), performing 30 independent runs yield good estimates (and an intensive use of the mentioned machines). We will assess the degree of significance of the tests by computing Student's t-test values (as will be explained in Section 5).

4.3. Heterogeneous Computing Resources

A heterogeneous network is one in which multiple, possibly dissimilar, machines cooperate in solving a problem. The technology for executing programs on a heterogeneous network of machines has advanced to a point where it can become an increasingly useful tool for solving large problems. A heterogeneous computing network may provide significant speedups over homogeneous systems in many cases, even super-linear speedups. In some cases, particularly efficient algorithms should execute faster on the heterogeneous network of m machines than they would on m copies of the same fastest machine [9].

Our heterogeneous computing system is made up of clusters of machines in which most modern and usual operating systems can be found. In this paper, we are reporting results on:

1. a cluster of Digital AlphaServers (four nodes with four processor each, connected by proprietary Memory Channel technology [11] and also by Fast Ethernet),
2. a cluster of 8 personal computers running Windows NT WorkStation 4.0,
3. a SGI OCTANE biprocessor machine running IRIX 6.5, and
4. a personal computer running both Windows NT 4.0 and Red Hat Linux 6.2.

See a graphical sketch of the heterogeneous system in Fig. 4. Table 1 contains the details about clock speed, memory, operating system, Java version, etc.

We will provide first an analysis on different sets of homogeneous machines and then tackle the heterogeneous case, once the homogeneous one has been addressed. For all the tests, we inform of the number of processors being used (`1p`, `4p`, etc.). When using the AlphaServer further information is required to characterize the results. In particular, we need to report the number of processors and which of the 4 available nodes (with 4 processors each) they are selected from. Also, we must report whether Fast Ethernet (FE or `fe`) or Memory Channel (MC or `mc`) is being used as the communication network linking the processors. Thus, `8p-4n-mc` means that we are using 8 processors and 4 nodes (hence we are using 2

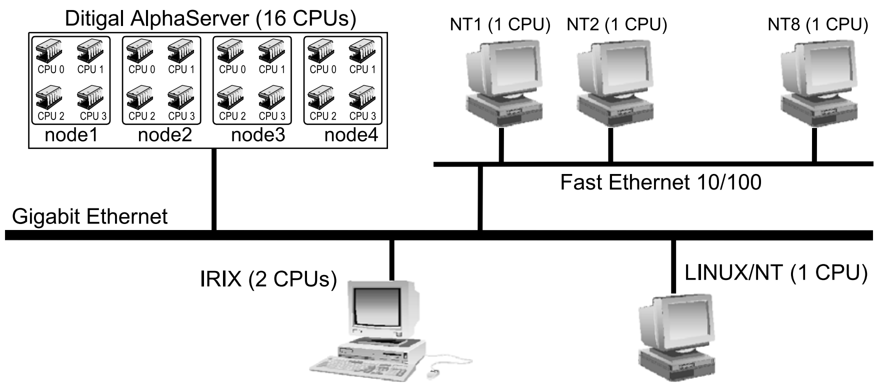

FIG. 4. Heterogeneous cluster of machines.

TABLE 1

Heterogeneous Computing Systems

[DIGITAL] Digital Alpha Server Cluster	
System:	Digital UNIX V4.0D (Rev. 878)
	DIGITAL UNIX Spanish Support V4.0D (Rev. 16)
Features:	AlphaServer 4100 PBXDA-AA RISC
	4 available AlphaServer each one with 4 processors at 300 MHz and 256 MB RAM
	Memory Channel available
	Fast Ethernet available
Java:	JDK 1.2.1, jit
[NTc] Personal Computer Cluster	
System:	Windows NT 4 (Workstation)
Features:	Pentium III, MMX Intel at 550 MHz with 128 MB RAM
Java:	JDK 1.2.2-001, symcjit
[IRIX] Biprocessor	
System:	IRIX
Features:	SGI OCTANE, MIPS R10000 (IP30) with 256 MB RAM
	IRIX-64 Release 6.5
Java:	JDK 1.2.2, mipsjit
[LINUX] [NT] Personal Computer	
System:	Windows NT 4 (Workstation)
	Linux kernel 2.2.14-5.0 Red Hat 6.2
Features:	Pentium III, MMX Intel at 500 MHz with 128 MB RAM
Java:	JDK 1.2.2 javacomp (Linux)
	JDK 1.2.2-001, symcjit (NT)

processors of every node leaving other two processors unused in each node) linked by Memory Channel. (Refer to Table 2 for learning more about the nomenclature).

TABLE 2

Nomenclature

m	Number of processors
t(*s*)	Time in seconds
s	Speedup
sf	Serial fraction
t-test	Student's *t*-test
p-value	Significance value of a *t*-test
sps	Steps per second = number of steps/time
`fe`	Fast ethernet
`mc`	Memory channel interconnection network (shared memory)
`1p`, `2p`, `4p`, `8p`	1, 2, 4, 8 processors
`1n`, `2n`, `3n`, `4n`	1, 2, 3, 4 nodes of a Digital AlphaServer machine

5. TESTS AND ANALYSIS

In this section, we present the results of running the PGA on different hardware/software platforms for the problems just described in Section 4.1. We analyze the effort needed to locate a solution, the execution time, speedup, serial fraction, and we point out particular features of every execution environment relating the number of evaluations per second, and unexpected scenarios.

The discussion for every problem is structured in the following manner. First, the uniprocessor configuration is analyzed, and then the parallel homogeneous configurations are dealt with. After that, we study some special parallel configurations of the algorithm to show its dependencies from both algorithmic and running conditions (only for the first problem). Table 3 explains the mapping of the 8 islands to the ring of processors for the heterogeneous configurations. We discuss the results worked out by two different heterogeneous configurations; the first mapping places the islands consecutively in the set of processors detailed in the HET-1 row of Table 3, while the second heterogeneous configuration is described in the HET-2 row of this same table.

We have computed the associated *p*-values to decide the statistical significance of the results by a *t*-test. When making a *t*-test for two sets of 30 results A and B (e.g., two hardware configurations, two different algorithms, etc.) a *p*-value equal or below to 0.05 means that the two compared samples A and B are different with a 95% degree of confidence, thus indicating that one of them is better than the other. This helps us in concluding that one configuration is "more efficient" than another one, or in concluding that the differences in their means are not significant (and thus that A and B provide similar results).

We have attached to every row of the result tables a letter from the alphabet (`a`, `b`, `c`, ...) in order to better explain through references the performed *t*-tests and other results such as the heterogeneous speedup (in this last case to clarify which is the basic uniprocessor value used for speedup).

TABLE 3

Heterogeneous Configurations

Island id	1	2	3	4	5	6	7	8
HET-1	NT	NT	NT	NT	IRIX	LINUX	DIG(mc)	DIG(fe)
HET-2	NT	IRIX	NT	LINUX	NT	DIG(fe)	NT	DIG(fe)

5.1. *ONEMAX Problem*

Before beginning the discussion, we must remember the light weight of this function if we compare it with the grain of every algorithmic step. Although our operations are quite simple (linear complexity with the string size), computing the number of ones in one string is a very fast operation in any of the used platforms. Many interesting functions of larger difficulty have a computational grain similar to that of this problem. This is a two-fold advantage, since ONEMAX can serve to provide a baseline as well as informing of physical expectations relating number of steps per second and run times.

5.1.1. Uniprocessor configuration. Let us begin by analyzing the entries in Table 4 containing the results of execution the PGA on one single processor. The number of evaluations is clearly similar for IRIX, DIGITAL and LINUX, with and advantage for IRIX (rows a, c, d, k, p, q). The NT single processor execution (rows k and p), needs one order of magnitude more evaluations, which can be only explained by the random number generator and parallel scheduler of its Java implementation, since the PGA is exactly the same in all the tests. In fact, the number of evaluations is quite similar among the different parallel configurations of the NT cluster (rows k to o) clearly pointing out the larger number of steps needed in NT with respect the other operating systems.

As to the run time with one single processor (Table 5 rows a, c, d, k, p, q) the efficiency (from high to medium) is LINUX>IRIX>NT>DIGITAL. There are slight differences between using a single processor in the NT cluster and a desktop office NT (rows k and p) but without statistical significance (p-value $= 0.403$).

Therefore, the LINUX machine *with one processor* provides both a lower number of evaluations and the globally fastest run time. This allows us to forecast good performances for PGAs when run in clusters of LINUX machines (next step in our research). In addition, we will use this value for computing speedup in the heterogeneous case, since the LINUX machine provides the fastest uniprocessor execution, as discussed in Section 3.1.

5.1.2. Homogeneous multiprocessor configuration. In this subsection, we explain and analyze the results in the case of using a homogeneous cluster of similar workstations for running the PGA. Again, Tables 4 and 5 will be used for discussing the effort of finding a solution and the run time, respectively.

In the IRIX biprocessor, using 2 processors, each one running 4 islands, provides a surprising but significant increase (p-value $\ll 0.05$) in the number of evaluations. See

TABLE 4

Number of Evaluations for ONEMAX: Homogeneous and Heterogeneous Configurations

		m	Mean number of evaluations	(*t*-test) *p*-Value
IRIX				
	(a)	`1p`	77,205.3	(a,b) 0.006
	(b)	`2p`	83,206.9	
DIGITAL				
	(c)	`1p-1n-fe`	83,262.8	(c,d) 0.059
	(d)	`1p-1n-mc`	84,550.2	
	(e)	`4p-1n-f3`	85,050.1	(e,f) 0.691
	(f)	`4p-1n-mc`	85,460.6	
	(g)	`8p-2n-fe`	84,023.3	(g,h) 0.688
	(h)	`8p-2n-mc`	83,637.2	
	(i)	`8p-4n-fe`	85,303.7	(i,j) 0.089
	(j)	`8p-4n-mc`	83,800.7	
NTc				
	(k)	`1p`	840,878.3	(k,l) 0.887
	(l)	`4p`	836,349.6	(l,n) 0.515
	(m)	(mf = 512) 4p	672,419.1	(l,m) 2.3×10^{6}
	(n)	`8p`	814,510.8	(k,n) 0.401
	(o)	(backgr.) 8p	791,005.8	(n,o) 0.450
NT				
	(p)	`1p`	1,063,453.7	(k,p) 1.1×10^{7}
LINUX				
	(q)	`1p`	86,607.7	
HET-1				
	(r)	`8p`	249,104.9	
HET-2				(r,s) 3.4×10^{8}
	(s)	`8p`	360,948.2	

Table 4 to notice this fact. The increase from 77,205.3 to 83,206.9 occurs since any communication between islands, OS context changes, and termination always induce a larger time in which many additional evaluations are performed in the 2-processor case. If we take a look to the running time we will conclude that using one processor is faster than using two processors for the IRIX (!). This result is statistically significant (p-value $= 0.006$), and yields a bad speedup value (0.927) and the very large serial fraction 1.158, meaning that the grain of the algorithm is quite high in this platform and thus its parallelization for the ONEMAX with two processors is harmful (similar to panmixia but slower because of the communications).

The tests with the Digital AlphaServer work out a commonsense result in that all the configurations need a similar number of steps. If we analyze the total run time of the algorithm as a whole we do note differences. The speedup is quite good, not only linear but super-linear for 4 and 8 processors (column s in Table 5). Using Memory Channel (shared reflective memory for communications) is slightly better than using Fast Ethernet (rows `c`–`d`, `g`–`h`, and `i`–`j`). Also, using 4 nodes is more efficient than

TABLE 5

Run Times for ONEMAX: Homogeneous and Heterogeneous Configurations

		m	*t*(*s*)	*s*	*sf*	(*t*-test) *p*-Value	*sps*
IRIX							
	(a)	`1p`	124.3			(a,b) 0.006	621
	(b)	`2p`	134.1	0.927	1.158		620
DIGITAL							
	(c)	`1p-1n-fe`	490.8			(c,d) 0.001	169
	(d)	`1p-1n-mc`	476.7				177
	(e)	`4p-1n-fe`	119.1	4.121	−0.009	(e,f) 0.057	714
	(f)	`4p-1n-mc`	122.0	4.023	−0.001		700
	(g)	`8p-2n-fe`	66.4	7.417	0.011	(g,h) 10^{-4}	1269
	(h)	`8p-2n-mc`	62.2	7.890	0.002		1344
	(i)	`8p-4n-fe`	62.1	7.896	0.001	(i,j) 0.020	1372
	(j)	`8p-4n-mc`	60.6	8.092	−0.001		1381
NTc							
	(k)	`1p`	309.9			(k,l) $<2.2 \times 10^{-16}$	2712
	(l)	`4p`	77.9	3.976	0.002	(l,n) $<2.2 \times 10^{-16}$	10,730
	(m)	(mf = 512) `4p`	63.8	4.854	−0.058	(l,m) $<9.9 \times 10^{-16}$	10,531
	(n)	`8p`	38.5	8.051	−0.001	(n,k) $<2.2 \times 10^{-16}$	21,155
	(o)	(backgr.) `8p`	37.5	8.263	−0.004	(n,o) 0.500	21,087
NT							
	(p)	`1p`	300.3			(k,p) 0.403	3540
LINUX							
	(q)	`1p`	116.2				744
HET-1							
	(r)	`8p`	21.3	(q/r) 5.435	0.067		11,644
HET-2							
						(r,s) 6.3×10^{-9}	
	(s)	`8p`	31.6	(q/s) 3.679	0.167		11,421

using 2 complete nodes, both for FE and MC. The serial fraction is always very near 0 and almost constant, positive for near-linear and negative for super-linear speedup, indicating a good performance for this algorithm and problem on the AlphaServer (although it compares bad with the NT cluster).

The cluster of NT processors provides similar conclusions: very similar number of evaluations with any number of processors, and linear and super-linear speedups (rows `k`, `l`, `n`). All the results are statistically significant for the execution time. This cluster is much faster than the DIGITAL one, since the Java abstract machine implementation is much more optimized in NT than the available one for the Digital platform, and since our NT Pentium III processors perform much faster than our Digital ones. The configuration with 8 NT processors yields the overall faster execution (row `n`). Although a special scenario (row `o`, next section) seems to perform slightly faster, the *p*-value (0.5) indicates that the difference is not significant.

If we compare the steps per second (*sps* column in Table 5) we can see that the NT cluster provides the best values. The overall best homogeneous result for ONEMAX is achieved with 8 NT processors (21155 *sps*) and with 4 processors (10730 *sps*). As expected, the larger the computational power (i.e., the number of processors, memory, caché, etc.) the larger the *sps* is in any workstation cluster. This remarks the importance of using parallel computers in the evolutionary computing domain.

5.1.3. Special parallel configurations. In order to show that we are not biasing the results, we briefly present two tests performed with the NT cluster. The first scenario consists in changing the default migration gap from 32 to 512 with 4 processors, and the second one consists in executing every island in background in each NT workstation with 8 processors (rows `m` and `o`, respectively).

As expected, a larger migration gap can significantly reduce the numeric effort (from 836349.6 down to 672419.1) and of course the average run time (from 77.9 down to 63.8). Compare rows `l` and `m` in Tables 4 and 5. Interestingly, we note that the mean number of steps per second remains constant ($\sim 10^4$). It is a well-known fact that semi-isolated evolution allows faster solutions for many models of PGAs [3, 6, 30].

The background test (row `o`) is aimed to find out whether running in fore/background yields very different results, an interesting issue for NT users. We did not execute any spurious task at the same time in the processors, we just only run 30 executions in the background of processor-dedicated workstations. The result, although it seems to be slightly more efficient than the foreground (default) case (cf. rows `n` and `o`) does not have any statistical significance (p-value is $0.5 \gg 0.05$ for the run time). Our conclusion is then that the internal behavior of NT when a single task (our PGA) is running does not significantly differ whether the task is run foreground or background.

5.1.4. Heterogeneous multiprocessor configuration. We have tested two heterogeneous configurations of the machines. See again Table 3 for the details. Basically, our goal is to merge all the platforms to solve the same problem by using the same PGA. We expect a problem-dependent behavior since the heterogeneous execution is defined by a large number of parameters: different processor technologies, cachés, communication networks (in the case of HET-1 we use FE and MC), different memories, etc.

For the ONEMAX problem we are very pleased with the behavior of the heterogeneous configurations. They both have reduced spectacularly the number of necessary steps and the run time. See this result in Tables 4 and 5 (rows `r` and `s`). The best previous homogeneous results were achieved with the NT cluster, but the heterogeneous combination of 4 NT processors with the IRIX, LINUX and DIGITAL (HET-1) has improved the homogeneous results (needing on the average 21.3 s versus 38.5 s with 8 NT processors). The best result is worked out by HET-1 since the 4 NT workstations are used to place consecutive islands of the ring. We can confirm that the HET-2 configuration is slightly slower than the HET-1 configuration (p-value is small much below 0.05).

Since we compute the speedup with respect the fastest uniprocessor (the LINUX machine) the speedup is sub-linear (5.4 for HET-1 and 3.6 for HET-2). This is a good value. Although it might seem a poor result we can see that the serial fraction is very small, especially for HET-1, thus meaning that the speedup value corresponds to an efficient algorithm.

Our conclusion is then that the two heterogeneous executions, in this case, have provided very good measures of performance. Since there are many interesting functions with approximately the same grain as the ONEMAX (but with a more large difficulty and interest) these are good news.

We now offer a summary for the ONEMAX problem before proceeding with the second one. The most efficient uniprocessor execution is achieved with a LINUX machine. The number of necessary evaluations is very similar in all the cases for equivalent homogeneous hardware, although the numeric effort changes from one platform to another. The main reason for this is the implementation of Java. We report very good values of speedup (linear and near-linear) and even super-linear speedup is often encountered, thus confirming this super-performance scenario both for shared memory multiprocessors and distributed memory multi-computers. Caché effects and the parallel search of the problem space are here the sources for this behavior. We cannot, however, discard the implementation source for super-linear speedup, since although our algorithm is the same in all the cases, the Java abstract machine is different for each platform. Heterogeneous computing is a way of taking profit for existing networks of workstations, and at the same time, this can even be more efficient than using an equivalent network of similar number of (even faster) computers.

5.2. *P-PEAKS Problem*

In this section, we discuss the results obtained when solving the P-PEAKS problem with three different configurations: uniprocessor, homogeneous multiprocessor, and heterogeneous multiprocessor. Tables 6 and 7 contain all the details for the forthcoming discussion. We are using an instance of this problem generator of high complexity and with large computational demands, since we are using strings of 512 bits and also we are considering 512 peaks. This means that *every evaluation* of one string needs to compute Hamming distance and several other minor operations over this set of 512 peaks. Our aim is to check whether some of our precedent findings still hold and what are the new considerations for such a problem.

Although for this problem we do not perform the two special scenarios reported in rows `m` and `o`, we keep the same labeling as for ONEMAX for consistency.

5.2.1. Uniprocessor configuration. If we analyze the single processor cases (rows `a`, `c`, `d`, `k`, `p`, `q`) we can see that DIGITAL, NT, and LINUX machines spent a similar number of evaluations (with very small differences among them). This is still the same result that we got for ONEMAX (except for NT), although absolute figures are not so different each other as in the ONEMAX case, since every step is quite time consuming in P-PEAKS.

TABLE 6

Number of Evaluations for P-PEAKS: Homogeneous and Heterogeneous Configurations

		m	Mean number of evaluations	(t-test) p-Value
IRIX				
	(a)	`1p`	75,151.0	(a,b) 0.0687
	(b)	`2p`	76,458.8	
DIGITAL				
	(c)	`1p-1n-fe`	96,120.8	(c,d) 0.385
	(d)	`1p-1n-mc`	97,345.3	
	(e)	`4p-1n-f3`	94,666.2	(e,f) 0.498
	(f)	`4p-1n-mc`	95,828.7	
	(g)	`8p-2n-fe`	92,419.2	(g,h) 0.188
	(h)	`8p-2n-mc`	94,563.0	
	(i)	`8p-4n-fe`	95,871.6	(i,j) 0.013
	(j)	`8p-4n-mc`	91,987.8	
NTc				
	(k)	`1p`	98,276.9	(k,l) 0.114
	(l)	`4p`	96,703.6	(l,n) 0.642
	(n)	`8p`	97,279.0	(k,n) 0.414
NT				
	(p)	`1p`	95,603.1	(k,p) 0.015
LINUX				
	(q)	`1p`	96,732.0	
HET-1				
	(r)	`8p`	58,166.1	(n,r) $<2.2 \times 10^{-16}$
HET-2				(r,s) 0.485
	(s)	`8p`	58,852.4	(n,s) $<2.2 \times 10^{-16}$

The IRIX machine is quite efficient in reducing the number of computations to solve the problem. Only 75151 evaluations is really a good uniprocessor score for this problem (with our algorithm and parameters).

From the point of view of the run time we can see that a single NT processor (rows `k` and `p`) is the fastest configuration (40 *sps*), followed by the LINUX, IRIX, and finally the DIGITAL uniprocessors (this last case with a run time one order of magnitude larger). In this case, the resulting ranking match the underlying technologies' state of the art, since the NT/LINUX processors are the most modern ones in our machine set, followed by IRIX and DIGITAL. The conclusion is then that for time demanding applications the underlying hardware is a very important factor.

5.2.2. Homogeneous multiprocessor configuration. In the case of homogeneous clusters we have many outcomes. Again, for the IRIX, the result in evaluations and time is worse for 2 processors than for one single processor (Tables 6 and 7, rows `a` and `b`). However, this time the t-test indicates that their differences in evaluations

TABLE 7

Run Times for P-PEAKS: Homogeneous and Heterogeneous Configurations

		m	$t(s)$	s	sf	(t-test) p-Value	sps
IRIX							
	(a)	`1p`	4153.3			(a,b) 0.711	18
	(b)	`2p`	4219.0	0.984	1.031		18
DIGITAL							
	(c)	`1p-1n-fe`	27,086.8			(c,d) 0.289	3
	(d)	`1p-1n-mc`	28,166.2				3
	(e)	`4p-1n-fe`	7719.4	3.509	0.046	(e,f) 9.7×10^{-4}	12
	(f)	`4p-1n-mc`	8297.1	3.394	0.059		11
	(g)	`8p-2n-fe`	3859.4	7.018	0.020	(g,h) 0.026	24
	(h)	`8p-2n-mc`	4036.0	6.978	0.020		25
	(i)	`8p-4n-fe`	3927.9	6.896	0.022	(i,j) 0.237	24
	(j)	`8p-4n-mc`	4005.9	7.031	0.019		23
NTc							
	(k)	`1p`	2425.4			(k,l) $<2.2 \times 10^{-16}$	40
	(l)	`4p`	604.1	4.014	−0.001	(l,n) $<2.2 \times 10^{-16}$	160
	(n)	`8p`	300	8.083	−0.001	(n,k) $<2.2 \times 10^{-16}$	324
NT							
	(p)	`1p`	2439.2			(k,p) 0.620	39
LINUX							
	(q)	`1p`	4103.7				23
HET-1							
	(r)	`8p`	316.8	(k/r) 7.655	0.006	(n,r) 0.001	183
HET-2						(r,s) 0.308	
	(s)	`8p`	322.0	(k/s) 7.532	0.008	(n,s) 4.0×10^{-16}	182

and time are not significant. In fact, the speedup and the serial fraction have undesirable values (Table 7, rows `a` and `b`).

Also for this second problem, the DIGITAL multiprocessor works out quite high computing times for any number of processors and configurations, even worse than using a single NT processor (!). We guess that the reason is twofold: the relatively slow processors, and a slow implementation of Java. However, if we take account for the DIGITAL configurations by themselves, we can see that speedup is near-linear (at a very small and almost constant serial fraction) consequently indicating a good performance of the algorithm.

We also get for this problem the same conclusion that for ONEMAX: the homogeneous cluster of NT workstations is the fastest one in solving the problem (see e.g., row `n` in Table 7: 300 s and 324 *sps*).

The NT cluster shows meaningful results (Table 7, rows `k`, `l`, `n`) in that the running time is consistently reduced as more processors are added. In fact, super-linear speedup is achieved at a very small serial fraction, which is an indication of very efficient performance. The numeric effort is basically the same for 1, 4 and 8 processors, since the differences are not statistically significant, therefore indicating that the NT machines keep the basic semantic of the algorithm.

We can compare the computational grain of ONEMAX and P-PEAKS through their associated steps per second column (Tables 5 and 7). We can see that the results for the P-PEAKS are always below (sometimes much below) 1000 steps per second. However, for ONEMAX we usually had values considerably larger (around 10,000 or 20,000 steps per second).

5.2.3. Heterogeneous multiprocessor configuration. The results with the two heterogeneous configurations (Tables 6 and 7, rows `r` and `s`) show that the number of evaluations is drastically reduced in both heterogeneous cases with respect the NT cluster with 8 processors and even the IRIX. This is probably due to the different rates and diversified zones of the problem that are searched in parallel by the different machines.

The heterogeneous results are quite good: speedup ~ 7.5 with 8 processors at very small serial fractions (see Table 7, rows `r` and `s`). In fact, this result is still better than for ONEMAX, and it compares quite well with the performance of the NT cluster.

We cannot say that HET-1 is better/worse than HET-2 since p-values (0.485 and 0.308 for evaluations and time, respectively) are far above 0.05. The heterogeneous configurations contain 'slow' machines such as two DIGITAL processors, and still their result is only somewhat slower than the NT cluster since they have very similar figures (compare rows `n`, `r`, and `s` in Table 7). However, strictly speaking, the 8 NT processors perform faster than both heterogeneous configurations: 300 s for the NTc versus 316.8 and 322.0 for HET-1 and HET-2, respectively. The difference is statistically significant (see p-values for `n`–`r` and `n`–`s`), although absolute times are very similar for NTc, HET-1, and HET-2.

Again, the conclusion is that reusing different machines can be as good as buying a whole cluster of similar machines. Having different machines is quite a good issue for labs in which users have different requirements; now we have shown that heterogeneity can be also very amenable for optimization with parallel GAs.

6. CONCLUDING REMARKS

In this paper, we have developed a PGA for solving optimization problems implemented in Java to get access both to homogeneous and heterogeneous networks of computers. Our main goals were to analyze such systems for evolutionary optimization methods, as well as extending some traditional-like studies to this heuristic non-deterministic field. We have compared the performance and behavior of this algorithm on two problems over a variety of hardware platforms. Our results indicate that LINUX is a good choice although only uniprocessor results are shown here for this platform. With respect to parallelism, the network of NT workstations has provided the overall best results.

Surprisingly, we have found a very significant reduction in the number of steps needed to solve the problem when using heterogeneous configurations. The reason can be found in the parallel exploitation of the search space at different rates and from different regions. Sometimes this could be a drawback in evolutionary algorithms, but we are very pleased since heterogeneity has been able to provide similar and even better results than homogeneous clusters. Since their history, budget, or idiosyncrasy, actual research labs have heterogeneous clusters; in this paper we have shown that connecting them can provide as good results as using similar machines.

We have experienced super-linear speedup for many configurations and problems, hence concluding that the island model is quite suited for modern clusters of workstations. Although super-linear speedup is always controversial, it is a research fact. Numerically speaking, an algorithm with structured population usually needs a smaller number of evaluations to find a solution, thus providing a faster execution. On the physical side, caché effects and the reduced complexity of the operations allow for super-linear speedup. Using the serial fraction measure helps in detecting subtle effects, as well as in assuring the quality of a given speedup value.

Heterogeneous computing means dealing with different hardware and software in its broadest sense, with a minimum requirement for the connectivity among the parts of the algorithm. Java has 'easily' solved the problems concerning operating systems, data representation, and data transmission, with a single set of programming tools. We know that using C++ can turn the applications several times faster than with Java, but this trend is rapidly changing nowadays, with the added advantage in Java of not needing additional communication libraries and its close relationship with the Internet.

Our future research must count necessarily for a further study with real-life applications. Metacomputing is also an interesting issue, although we want to maintain the discussion in the algorithmic terrain, which is difficult when using the present metacomputing systems such as Globus [10]. Finally, evaluating the influence of using a richer set of communication networks including ATM, Gigabit Ethernet, Myrinet, and others, is an important issue we plan to tackle in the near future.

ACKNOWLEDGMENT

This work is partially supported by the National CICYT Project TIC1999-0754-C03-03.

REFERENCES

1. S. G. Akl, "The Design and Analysis of Parallel Algorithms," Prentice-Hall, Englewood Cliffs, NJ, 1991.
2. E. Alba and J. M. Troya, A survey of parallel distributed genetic algorithms, *Complexity* **4** (1999), 31–52.
3. E. Alba and J. M. Troya, Analyzing synchronous and asynchronous parallel distributed genetic algorithms, *Future Generation Comput. Systems* **17** (2001), 451–465.
4. D. Andre and J. R. Koza, A parallel implementation of genetic programming that achieves super-linear performance, *J. Inform. Sci.* **106** (1998), 201–218.
5. T. Bäck, "Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms," Oxford Univ. Press, New York, 1996.

6. T. C. Belding, The distributed genetic algorithm revisited, *in* "Proceedings of the Sixth International Conference on Genetic Algorithms" (L. J. Eshelman, Ed.), pp. 114–121, Morgan Kaufmann, Los Altos, CA, 1995.
7. P. Calégari, F. Guidec, P. Kuonen, and D. Kobler, Parallel island-based genetic algorithm for radio network design. *J. Parallel Distrib. Comput.* **47** (1997), 86–90.
8. E. Cantú-Paz and D. E. Goldberg, Predicting speedups of idealized bounding cases of parallel genetic algorithms, *in* "Proceedings of the Seventh International Conference on Genetic Algorithms" (T. Bäck, Ed.), pp. 113–120, Morgan Kaufmann, Los Altos, CA, 1997.
9. V. Donaldson, F. Berman, and R. Paturi, Program speedup in a heterogeneous computing network, *J. Parallel Distrib. Comput.* **21** (1994), 316–322.
10. I. Foster and C. Kesselmann, Globus: A metacomputing infrastructure toolkit, *J. Supercomput. Appl.* **11** (1997), 115–128.
11. R. Gillet, Memory channel network for PCI, *IEEE Micro.* **16** (1996), 12–18.
12. D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning," Addison–Wesley, Reading, MA, 1989.
13. V. S. Gordon and D. Whitley, Serial and parallel genetic algorithms as function optimizers, *in* "Proceedings of the Fifth International Conference on Genetic Algorithms" (S. Forrest, Ed.), pp. 177–183, Morgan Kaufmann, Los Altos, CA, 1993.
14. UEA CALMA Group, "Calma Project Report 2.4: Parallelism in Combinatorial Optimisation" Technical Report, School of Information Systems, University of East Anglia, Norwich, U.K., September 18, 1995.
15. J. L. Gustafson, G. R. Montry, and R. E. Brenner, Development of parallel methods for a 1024-processor hypercube, *SIAM Sci. Statist. Comput.* **9** (1988), 609–638.
16. W. E. Hart, S. Baden, R. K. Belew, and S. Kohn, Analysis of the numerical effects of parallelism on a parallel genetic algorithm, *in* "IEEE Proceedings of the Workshop on Solving Combinatorial Optimization Problems in Parallel" (IEEE, Ed.), pp. CD-ROM IPPS97, IEEE Press, New York, 1997.
17. J. H. Holland, "Adaptation in Natural and Artificial Systems," The MIT Press, Cambridge, MA, 2nd ed., 1992.
18. K. A. De Jong, M. A. Potter, and W. M. Spears, Using problem generators to explore the effects of epistasis, *in* "Proceedings of the 7th International Conference of Genetic Algorithms" (T. Bäck, Ed.), pp. 338–345, Morgan Kaufman, Los Altos, CA, 1997.
19. A. H. Karp and H. P. Flatt, Measuring parallel processor performance, *Comm. ACM* **33** (1990), 539–543.
20. T. Lai and S. Sahni, Anomalies in parallel branch-and-bound algorithms, *Comm. ACM* **27** (1984), 594–602.
21. S. C. Lin, W. F. Punch, and E. D. Goodman, Coarse-grain parallel genetic algorithms: Categorization and a new approach, *in Sixth IEEE SPDP*, pp. 28–37, 1994.
22. Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs," 3rd ed., Springer-Verlag, Heidelberg, 1996.
23. H. Mühlenbein, M. Schomish, and J. Born, The parallel genetic algorithm as a function optimizer, *Parallel Comput.* **17** (1991), 619–632.
24. M. Munetomo, Y. Takai, and Y. Sato, An efficient migration scheme for subpopulation-based ansynchronously parallel genetic algorithms, *in* "Proceedings of the Fifth International Conference on Genetic Algorithms" (S. Forrest, Ed.), pp. 649, Morgan Kaufmann Publishers, San Mateo, CA, 1993.
25. J. Schaffer, L. J. Eshelman, and D. Offutt, Spurious correlations and premature convergence in genetic algorithms, *in* "Foundations of Genetic Algorithms" (G. J. E. Rawlings, Ed.), pp. 102–122, Morgan Kaufman, Los Altos, CA, 1991.
26. R. Shonkwiler, Parallel genetic algorithms, *in* "Proceedings of the Fifth International Conference on Genetic Algorithms" (S. Forrest, Ed.), pp. 199–205, Morgan Kaufmann, Los Altos, CA, 1993.

27. P. Spiessens and B. Manderick, A massively parallel genetic algorithm, *in* "Proceedings of the Fourth International Conference on Genetic Algorithms" (L. B. Booker and R. K. Belew, Eds.), pp. 279–286, Morgan Kaufmann, Los Altos, CA, 1991.

28. J. Sprave, A unified model of non-panmictic population structures in evolutionary algorithms, *in* "Congress on Evolutionary Computation (CEC'99)," pp. 1384–1391, IEEE Press, Piscataway, NJ, 1999.

29. G. Syswerda, A study of reproduction in generational and steady-state genetic algorithms, *in* "Foundations of Genetic Algorithms" (G. J. E. Rawlins, Ed.), pp. 94–101, Morgan Kaufmann, Los Altos, CA, 1991.

30. R. Tanese, Distributed genetic algorithms *in* "Proceedings of the Third International Conference on Genetic Algorithms" (J. D. Schaffer, Ed.), pp. 434–439, Morgan Kaufmann, Los Altos, CA, 1989.

31. D. Whitley and T. Starkweather, GENITOR II: A distributed genetic algorithm, *J. Exp. Theoret. Artif. Intell.* **2** (1990), 189–214.

ENRIQUE ALBA has been working with evolutionary algorithms since 1991 in the department of "Lenguajes y Ciencias de la Computación" of the University of Málaga. He received his Ph.D. in Computer Science in 1999 from the University of Málaga (Spain) with a dissertation about designing and applying parallel genetic algorithms. At present he works as an associate professor in this department, and his current interests are evolutionary algorithms, parallel optimization algorithms, and their applications. He owns national and international awards to his research activities, and has authored many journal and conference papers on evolutionary algorithms.

ANTONIO J. NEBRO is an assistant professor at the University of Málaga (Spain). His current research interests deal with issues related to the implementation of concurrent object-oriented languages, and with the parallelization of optimization algorithms. He received his M.Sc. and Ph.D. degrees in Computer Science from the University of Málaga in 1991 and 1999, respectively.

JOSÉ M. TROYA received his Ph.D. in 1980 from the University Complutense of Madrid (Spain). From 1980 to 1988 he was an associate professor at that university, and since then he has been a professor at the Department of "Lenguajes y Ciencias de la Computación" at the University of Málaga. He has worked as the head of this department until 1998. His research interests are in parallel algorithms and software architectures, and in the development of languages and tools for programming and modeling distributed systems. He has directed many national and international research projects and numerous Ph.D. students in the mentioned fields.