Creation and Analysis of a JavaSpace-Based Distributed Genetic Algorithm

Brian Zorman, Gregory M. Kapfhammer, and Robert Roos

Abstract—**The island model for distributed genetic algorithms (GAs) is a natural match for the master-worker paradigm in distributed computation. We explore the benefits and drawbacks of several distributed system architectures in developing an implementation of a distributed GA that exploits the Jini and JavaSpace technologies. Our results, using the knapsack problem as an illustration, show that there is an unavoidable price to pay in terms of decreasing computation-to-communication ratios as a function of instance size. However, we can diminish these effects by expanding the number of JavaSpaces beyond those required for the obvious implementation. Our results also indicate that as the number of remote machines increases the potential for a better solution also rises. Even though our distributed GAs did not always exploit this potential for a higher quality solution, we believe that the combination of Java, Jini, and JavaSpaces presents avenues for easily distributing the computation of genetic algorithms.**

Keywords—**Distributed Systems, Genetic Algorithms, JavaSpaces, Jini**

I. Introduction

WHILE genetic algorithms (GAs) often perform better than more traditional search methods, they also have flaws, mainly expensive computation cost and poor solution quality due to premature convergence [12]. A GA executed in a single address space is likely to reach a point of equilibrium where offspring produced are very similar to their parents. This limited diversity causes the GA to explore only a confined area of the solution space, resulting in suboptimal solutions. A possible response to this problem is to create an environment where the GA can be run on several populations independently. In order to avoid early convergence, individuals from the independent populations can be merged with other populations in order to preserve diversity.

We present a new implementation of distributed genetic algorithms that uses Jini and JavaSpaces to improve the quality of the solution by distributing the initial population for execution across a range of machines. To demonstrate this, we add distribution functionality to an open source genetic algorithm for solving the knapsack problem. Then we design a distributed system model (DSM) using the Jini network technology and JavaSpaces object repositories [9], [18]. Jini is the backbone of the distributed system, and JavaSpaces provide a means of communication between machines in the system. We assess the quality of our design and implementation through a measurement of the computation-to-communication ratio, quality of solution, and diversity of the GA population. We compare the DSM to its sequential counterpart (SGA) through measurements of execution time and quality of solution.

In Section II we describe genetic algorithms, the Jini network technology, and the JavaSpaces object repository. Section III describes the design and implementation of our distributed system model and our genetic algorithm. Section IV introduces the knapsack problem and describes the metrics used in our evaluation procedure. In Section V, we present our experimental results and compare our results against those of the SGA.

II. Preliminaries

In this section we first give a brief description of some genetic algorithm terminology and concepts. Then we describe distributed systems, and conclude with details of the various technologies used.

A. Genetic Algorithms

GAs are search heuristics for combinatorial optimization problems [12]. They are best used in searching irregular problem spaces. In a GA, there are a number of elements called *individuals* or *chromosomes* that are made up of values called *genes*. Each individual is assigned a *fitness*. Individuals are grouped in a set of solutions called a *population* which evolves through many *generations*. During each generation, randomly selected individuals are combined and modified through a *crossover* operation to produce *offspring*. With some predefined probability, a *mutation* operation is performed on the offspring. If the offspring's fitness is above a specified threshold, it replaces an individual with a lower fitness. This process is repeated until some termination condition is achieved. A typical structure for a genetic algorithm is shown in Algorithm 1.

Algorithm 1 Genetic Algorithm Procedure

```
procedure GA;
start
Create initial population;
repeat
   Gather 2 individuals for crossover;
   Apply crossover to individuals gathered;
   Apply mutation according to mutation probability;
   Update population according to fitness;
until Reach termination condition;
end GA
```

B. Distributed Systems and GAs

A distributed system can be defined as a system where the information processing is distributed over several computers rather than confined to a single machine. Coulouris et al., Sommerville, and Tanenbaum et al. [6], [22], [24] have characterized distributed systems through consideration of the following properties: resource sharing, openness, concurrency, scalability, fault tolerance, and transparency.

Department of Computer Science, Allegheny College, Meadville, PA. E-mail: {zormanb,gkapfham,rroos}@allegheny.edu

If we add a *migration* operation to the standard GA described in Algorithm 1, then it could be executed in a distributed environment. Migration consists of selecting individuals from the GA population, sending them to another GA population which is evolving in parallel, and then receiving different individuals back from the other GA population. This type of resource sharing can lead to a decrease in the execution time of a genetic algorithm and an increase in the overall quality of solution. We describe such a migration operation in Section III.

There have been several distributed implementations of genetic algorithms [1], [4], [15], [19], [21], [23]. The most relevant to this paper discusses the *Jinetic* tool created by Atienza et al. [1]. Jinetic uses Jini network technology (but not JavaSpaces) to build a distributed GA. Jinetic has each genetic operation and even chromosomes set up as Jini services. This design choice resulted in a large amount of communication overhead. We introduce JavaSpace object repositories to provide for a more efficient and elegant solution, since they facilitate loosely coupled communication between entities in a Jini federation [9].

C. Technologies

The technologies used in this paper are the Java-based Jini network technology and JavaSpaces object repositories which were developed to address the challenges posed by distributed systems: complexity, security, manageability, and unpredictability [9], [24].

C.1 Jini network technology

Jini is a set of application programming interfaces (APIs) and runtime conventions that facilitate the creation of distributed systems. The entities within these Jini-based distributed systems communicate by Jini protocols. One way to facilitate this communication is through Java Remote Method Invocation (RMI) [13], [16]. RMI facilitates communication between remote machines by allowing references to objects to be held in other Java virtual machines (JVM). Through these references, methods can be executed on these objects and objects can be passed to these methods. The major components of a Jini system are [18]:

- *Jini lookup service* (JLUS): The central organizing mechanism for a Jini-based system. Services register with the lookup service in order to make themselves known. Clients query the lookup service in search of services they need.
- *Jini client*: An entity that can look up and retrieve a registered service and invoke methods of the service.
- *Jini service*: An entity containing methods that may be of use to some other Jini client or service, and that registers with lookup services to provide access to those methods.
- *Jini federation* : A collection of clients and services all communicating through Jini protocols.

The combination of Jini and Java can turn a network of heterogeneous computing entities into a homogeneous collection of clients and services that are associated with Java virtual machines. Jini reduces the complexity of distributed systems by providing elegant ways to handle distribution concerns. Through the usage of the Jini lookup service, it is easy for a client to locate services that fit its particular needs. Jini also provides an interface for interacting with the network topology, which hides low-level details from the user. For a more detailed discussion of the Jini network technology, please refer to [3] and [18].

C.2 JavaSpaces

In order to effectively handle the communication between machines in a distributed environment, a Jini service called JavaSpaces has been developed [9]. JavaSpaces not only aid in the design of distributed environments, but also offer a mechanism for persistence within a distributed environment. In other words, an object can remain in a JavaSpace for any specified length of time, even through system restarts. The JavaSpaces object repository also provides a high-level coordination mechanism for distributed systems by exploiting the familiar notion of a shared memory [9].

JavaSpaces builds on the Jini substrate. It takes the form of a Jini service within a federation and uses the Java features of remote method invocation (RMI) and serialization to pass parameters between remote machines. While RMI gives the power of sending objects across a network, serialization is used to ensure that the objects passed across the network are in a form that can be executed on a remote JVM [8].

JavaSpaces allow for services to communicate in a fashion that does not require the direct remote service method invocation. The JLUS facilitates the search that a Jini service must conduct in order to find an appropriate JavaSpace. Any system can communicate with a JavaSpace, from anywhere, and at anytime, as long as that system adheres to the protocols, making it ideal for loosely coupled services [9]. JavaSpaces was designed to be both simple and expressive. Its repository was designed to hold a typed group of objects called `Entry`. The simple lookup operations are `read`, `take`, and `write`. A `read` will return an `Entry` from the JavaSpace that matches the template provided by the requesting party. A `take` will return an entry from the JavaSpace, just as in a `read`, and also remove that `Entry` from the JavaSpace. A `write` will simply write an `Entry` to the JavaSpace [9]. Figure 1 describes the operations that are provided by the JavaSpaces object repository.

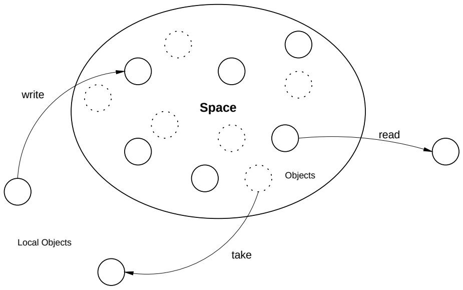

Fig. 1. An overview of JavaSpaces operations: `read`, `write`, and `take`.

III. Distributed System Design

The *island model* was chosen as the basis for our distributed system design [7]. In the island model, the initial population is evenly divided among a number of subpopulations (islands).

Each island executes all of the genetic operations, including a *migration* operation. Migration allows for diversity among the islands and prevents premature convergence. Also, there is a scalability benefit to using this model because an increase in the number of islands will lead to a larger diversity and a better solution [5], [7]. The drawbacks of the island model can be seen in a high communication cost depending on the frequency of migration and also the network structure used in the model [5], [7].

A. Simple Distributed System Model

The simple JavaSpace-based distributed system model (SDSM) is a variation of the island model that also follows the *master-worker* distributed systems paradigm [7], [24]. In this paradigm the master, or *initial machine*, acts as the control entity by populating the workers, or *remote machines* (islands), and receiving the results. The workers simply wait until they are given something to compute. Each worker is populated with identical-sized sets of individuals. At this point the workers can evolve their own initial population.

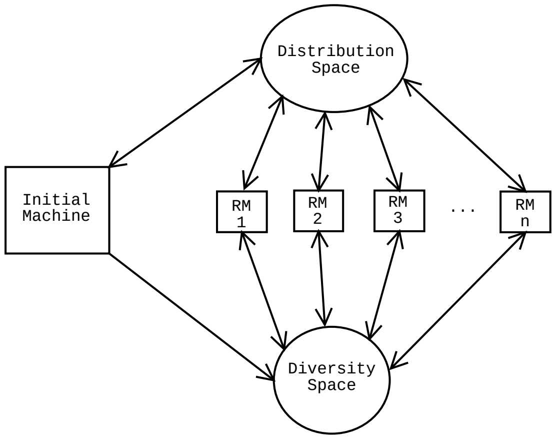

Fig. 2. Simple Distributed System Model using Jini and JavaSpaces

Figure 2 provides a high level view of the SDSM architecture. The SDSM uses two JavaSpaces (DistributionSpace and DiversitySpace), one initial machine, and n remote machines. The DistributionSpace is used for the distribution of the initial population over a number of machines. The DiversitySpace is used to handle communication between the remote machines during migration. In addition to populating the remote machines, the initial machine is also responsible for populating the DiversitySpace with individuals used in the first migration.

A.1 Drawbacks

Some performance concerns arise when taking a close look at the SDSM and the procedures that execute within it. As the GA instances grow in size, the sizes of the individuals in the population will also increase, leading to larger packages being sent back and forth from the DiversitySpace during migration. During our experimentation a slight increase in communication time was seen due to the increase in packet sizes being sent across the network.

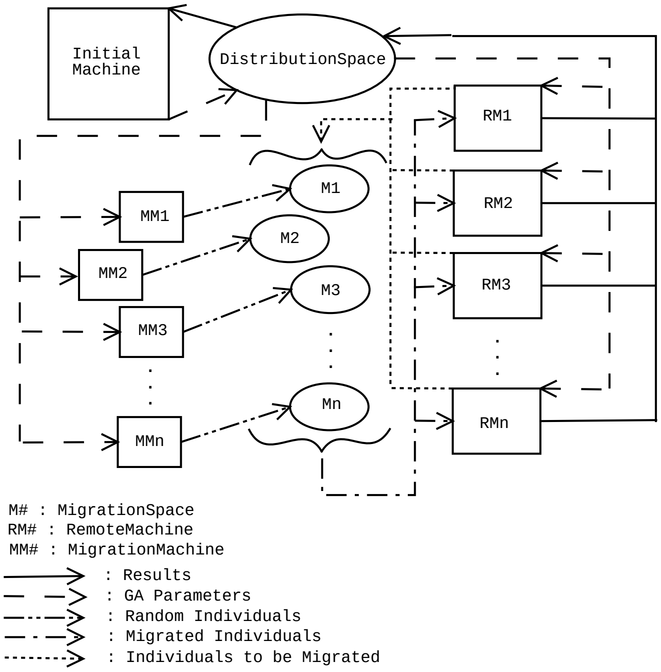

Fig. 3. Complex JavaSpace-Based Distributed System Model for the DGA

Without explicit synchronization, it is easy to see that the SDSM allows for all of the remote machines to initiate the migration procedure at the same time with the same DiversitySpace. A performance bottleneck occurs in this situation. The most obvious reason is the extra amount of network load on the one machine that hosts the DiversitySpace. Another cause could be an overwhelming number of attempted interactions with this JavaSpace. As the number of remote machines within the system model rises, the performance of the JavaSpace falls.

Finally, as all of the remote machines are trying to migrate with the JavaSpace, there is no computation within the entire system. The computation-to-communication ratio for the SDSM could fall to inadequate levels and possibly drop to the point where a sequentially executed genetic algorithm will have better performance. This situation has disastrous performance implications, and happened often enough to render the SDSM unacceptable. A more complex system could help eliminate some of these performance concerns.

B. Complex Distributed System Model

The complex distributed system model (CDSM) also follows a variation of the island model, but some other modules have been added to address performance problems, improve the computation-to-communication ratio and eliminate the possible bottleneck in the SDSM. Instead of having a single DiversitySpace for migration, this system has multiple *MigrationSpaces*. Each MigrationSpace also has its own *MigrationMachine*, whose job is to fill the MigrationSpace with the initial individual packages for early migrations. The initial machine and remote machines will have behaviors similar to those described for the SDSM.

The CDSM in Figure 3 makes use of one initial machine and n JavaSpaces, remote machines, and migration machines. Its execution begins at the initial machine. The only differences are that there is no longer a single DiversitySpace and the population packages sent by the initial machine are picked up by the remote machines as well as the migration machines.

The remote machines begin execution of the DGA following an algorithm similar to that described for the SDSM. When a remote machine reaches migration, it performs the first migration procedure with the unique MigrationSpace that was assigned by the initial machine. Subsequent migrations take place with the next MigrationSpace in the list. This synchronization mechanism is used in order to reduce the load put on a single JavaSpace by not allowing a large number of remote machines to access it at the same time. The synchronization process will result in all of the remote machines migrating with a different JavaSpace at any given time. This will reduce diversity of the populations on each remote machine since each MigrationSpace will not necessarily consist of individuals from all of the remote machines. To handle the initial migration, the migration machines will initially fill the MigrationSpaces with random packages.

As in the SDSM, upon termination of the GA running on the remote machines, the results are sent back to the DistributionSpace. The initial machine takes the results and compiles them.

IV. Experiment

We chose to adapt an existing Java-based solution to the knapsack problem to facilitiate a comparison between the SDSM, CDSM, and the sequential GA. For a detailed review of the weaknesses of the SDSM, please refer to [25]. We first define the knapsack problem and the GA that attempts to solve it. Then we discuss our testbench and the various metrics used for analysis.

A. Knapsack Problem

We employ a genetic algorithm [10] that provides a solution to the knapsack problem for the analysis of our simple distributed system model (SDSM) and complex distributed system model (CDSM), and their comparison to the sequential counterpart.

In the knapsack problem, we are given a set of n items, $S = \{1, \dots, n\}$, where item i has size s_i and value v_i. The knapsack has capacity C. We want to find the subset $S' \subset S$ that maximizes the value of $\sum_{i \in S'} v_i$, given that $\sum_{i \in S'} s_i \leq C$.

B. Knapsack GA

The genetic algorithm that solves this version of the knapsack problem executes in the following manner. At the start a set of solution points is randomly selected. (A solution point is a subset of S, represented as a bit string of length n, where the i-th bit is 1 if and only if the i-th item is selected for inclusion in the knapsack.) At each generation, the solution points are evaluated for fitness (according to how much of the knapsack capacity they fill), and the best and worst performers are identified. When conditions are met for crossover, the best solution mates with a random (non-extreme) solution, and the offspring replaces the worst one. A random solution may also be mutated to vary the gene pool [10].

C. Testbench

Our testbench consists of a set of solved knapsack instances. These problem instances were developed through use of a knapsack instance generator available on the OR Library [20]. We chose to create our testbench using item sets of sizes $\{500, 600, 700, 800, 900\}$; the value of each weight was in the range 0–5000. Through preliminary test-executions of the sequential GA, it was found that when using fewer than 500 items, very good solutions were always found. On the other hand, when the number of knapsack items exceeded 500, the number of good solutions decreased drastically. In Section V, we label harder test sets with higher numbers.

Along with the problem instances, the following GA parameters will remain constant as follows:

- *termination condition*: GA terminates when best solution is not updated after 75 generations
- *crossover rate*: 1.0, i.e., perform crossover at every generation
- *mutation rate*: 1.0, i.e., mutate at every crossover
- *migration rate* (SDSM and CDSM only): .30/30, i.e., migrate 30% of the population every 30 generations

In order to test the design of our system model, other parameters were varied in an organized way throughout the testbench. These include the number of remote machines for testing the CDSM, the number of migration machines for testing the CDSM, and the number of migration spaces for testing the CDSM. Each generated problem instance was executed using n remote machines for all $n \in \{2, 4, 6, 8, 10\}$. For the CDSM, the number of migration machines and migration spaces equalled the number of remote machines for each problem instance.

D. Metrics

Many of our measurements were time related. Execution time is the time from start to finish of the test run. Since GAs are known to have high execution costs, and distributed systems can provide for a reduction in those costs, execution time provides a logical means of comparison between the CDSM and the SGA.

In a distributed system, the higher the computation-to-communication ratio, the more efficiently its resources are used. Communication time is measured by "wrapping" every method call that accesses the JavaSpaces, i.e., `write` and `take`. These communication times are gathered for each remote machine in order to provide a specific remote machine-oriented evaluation. The computation-to-communication ratio is an average of the computation-to-communication ratios of all the remote machines.

Another means of evaluating the design of the SDSM and the CDSM is by measuring the diversity of the populations being developed. We define the diversity of a population, measured every ten generations, by subtracting the fitness of the least fit individual from that of the most fit individual. An average diversity for each remote machine, and also a total average diversity is computed for the entire system test run.

Solution quality is computed by taking the best solution as a percentage of the knapsack capacity designated for that test run.

V. Experimental Results and Discussion

The aim of our experiments was to not only compare the CDSM to its sequentially executed counterpart in terms of quality of solution and execution time, but also evaluate the design of our CDSM. Table I shows the results we gathered based on the metrics described in Section IV-D. In terms of quality of solution, the CDSM displayed an improvement over the SGA. As shown in Figures 4 and 5, a pattern begins to form in terms of the quality of solution. The quality of solution for the SGA gradually decreases throughout the tests, deteriorating to a low of 64% on the largest test set. On the other hand, the quality of solution for the CDSM outperformed the SGA throughout the test suite and remained roughly at 75%, a fact that surely can be attributed to the high diversity achieved through migration.

TABLE I
Performance of CDSM on the Testbench

Set	#Mach.	Exec. time	C/C	Diversity	QofS
1	seq	101009	n/a	589861.8	89.4
	2	181102	2.01	653249.4	84.4
	4	183860	1.39	1188449.6	87.7
	6	318224	.70	1616393.6	89.3
	8	378824	.74	1869822.2	93.8
2	seq	86054	n/a	677697.2	67.5
	2	123321	2.28	939233.8	69.2
	4	297380	.88	1377099.1	73.1
	6	393355	.60	2016844.6	75.4
	8	373855	.44	2407615.8	71.0
3	seq	123508	n/a	833481.7	75.4
	2	155280	1.29	1133174.6	77.0
	4	308782	.78	1875170.8	79.4
	6	316706	.49	2352512.2	75.4
	8	470343	.41	2606187.2	74.6
4	seq	58467	n/a	863806.7	69.2
	2	359062	.78	1031281.7	75.8
	4	340019	.77	1985523.5	79.1
	6	495672	.40	2398799.8	73.8
	8	435787	.37	3612381.0	74.7
5	seq	50138	n/a	937859.0	64.5
	2	148137	1.05	1248085.5	73.7
	4	294307	.55	1918584.8	76.0
	6	598345	.32	2657823.5	72.9
	8	598532	.28	3539287.8	75.5

The high execution cost seen in Figure 4 can be attributed to the excessive amount of communication time experienced by the remote machines during migration. As the chromosome size grows throughout the test suite, so does the amount of data migrated, therefore the amount of communication time will rise. Table I shows that the average computation-to-communication ratio over all test sets falls as the number of remote machines rises. Over many instances the ratio decreased to a level below one, meaning that the communication time actually exceeded the computation time. This result shows an inefficient allocation of resources since the GAs executed by the remote workers spend more time performing migration than computing generations.

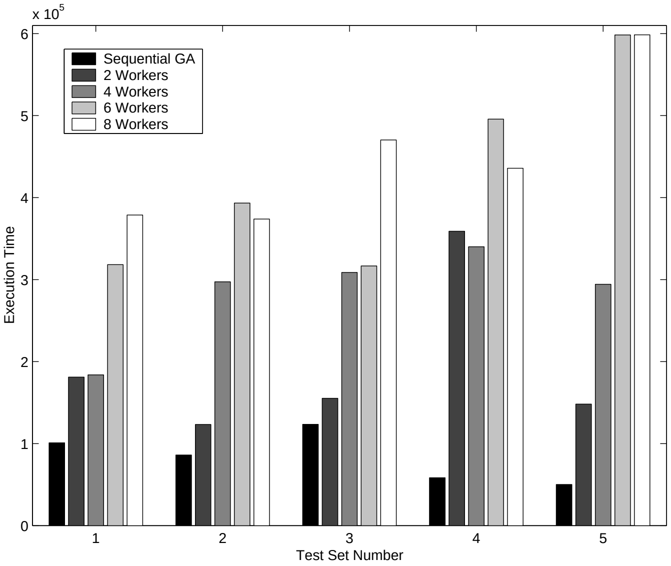

Fig. 4. CDSM compared to SGA in terms of execution time

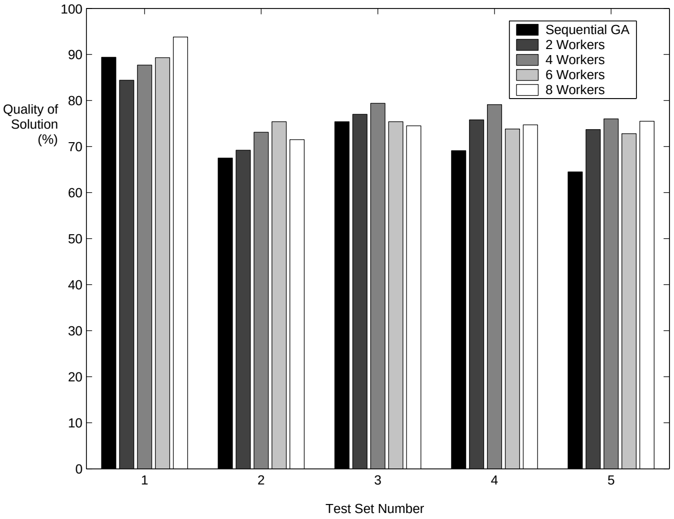

Fig. 5. CDSM compared to SGA in terms of quality of solution

However, a higher execution cost is actually desirable, as we want the GA in our CDSM to continue searching as long as it has a viable population. The SGA halts early due to stagnation of its population. Figure 4 shows that as the test set gets larger and more difficult, the execution time of the SGA diminishes.

Most of the improvements of the CDSM can be attributed to the high diversity achieved during execution. Table I shows that the diversity value computed for each test set grows as the number of remote machines grows. Therefore one can generally conclude that as the number of remote machines rises the *potential* for a better solution also rises. Yet, the distributed genetic algorithm does not always capitalize on this potential. Indeed, Figure 5 provides examples of test sets where an increase in the number of machines did not always increase the quality of solution.

VI. Conclusions and Future Work

This research project was undertaken to investigate the feasibility of using the Jini network technology and JavaSpaces object repositories for a distributed genetic algorithm that attempts to solve the knapsack problem. It was found that our CDSM out-performed the SGA in terms of quality of solution due to the high level of diversity achieved from our system design and the addition of the migration procedure to the GA. However, SGA had a better performance in terms of execution time. While some performance concerns can be attributed to the use of Jini and JavaSpaces [17], their simplicity and elegance as a backbone to a distributed system can provide for some interesting future research. For example, a framework could be developed in which any Java-based GA could be "plugged in" and executed. This opens the doors for testing a wide variety of NP-complete problems, such as traveling salesperson, multiprocessor scheduling problem, etc. Also, other JavaSpace implementations have been developed that improve upon their performance, such as GigaSpaces, RDBSpace, and GLOBE [2], [14], [11]. These might also be investigated in order to develop a more efficient distributed system for a genetic algorithm.

References

[1] J. Atienza, M. Garca, J. Gonzlez, and J.J. Merelo, Jinetic: a distributed, fine-grained, asynchronous evolutionary algorithm using Jini. `citeseer.nj.nec.com/atienza00jinetic.html`, January 2000.

[2] G. Arnold. Trading Spaces: Implementation and Analysis of a Relational Database JavaSpaces Service. Technical Report CS02-01, Department of Computer Science, Allegheny College, Meadville, PA. 2002.

[3] K. Arnold and B O'Sullivan and R. W. Scheifler and J. Waldo and A. Wollrath. *The Jini Specification*. Addison-Wesley, Inc, 1999.

[4] Theodore C. Belding. The distributed genetic algorithm revisited. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, San Francisco, CA, Morgan Kaufmann, 114–121, 1995.

[5] E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[6] J. Dollimore, T. Kindberg, and G. Coulouris. *Distributed Systems Concepts and Design*. Addison-Wesley, third edition, 2001.

[7] F. Ercal, A. Zomaya, and S. Olariu, editors. *Solutions to Parallel and Distributing Computing Problems*. John Wiley and Sons, 2001.

[8] D. Flanagan. *Java in a Nutshell*. O'Reilly and Associates, Inc, April 2002.

[9] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[10] A genetic knapsack problem solver. `http://members.aol.com/WindmiII96/knapsack/gks.html`.

[11] GigaSpaces Technologies Ltd. GigaSpaces Platform. `http://www.gigaspaces.com/index.htm`, 2002.

[12] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, New York, 1989.

[13] W. Grosso. *Java RMI*. O'Reilly and Associates, Inc., 2002.

[14] J. E. Larsen and J. H. Spring. GLOBE: A Dynamically Fault-tolerant and dynamically scalable distributed tuplespace for heterogeneous, loosely coupled networks. Ph.D. thesis, University of Copenhagen, Department of Computer Science. October, 1999.

[15] T. Matsumura, M. Nakamura, J. Okech, and K. Onaga. Parallel computation of distributed genetic algorithm on loosely-coupled multiprocessor systems. In *Proceedings of International Symposium on Artificial Life and Robotics*, pages 1–4, (1997-2).

[16] J. Newmarch. *A Programmer's Guide to Jini Technology*. APress, San Francisco, CA, 2001.

[17] M. Noble and S. Zlateva. Scientific Computation with JavaSpaces. In *Proceedings of Ninth International Conference on High Performance Computing and Networking*, pages 1–4, (June, 2001).

[18] S. Oaks and H. Wong. *Jini in a Nutshell*. O'Reilly and Associates, Inc, March 2000.

[19] J. Okech et al. A distributed genetic algorithm for the multiple knapsack problem using PVM. In *Proceedings of ITC-CSCC '96*, pages 899–902, 1996.

[20] David Pisinger. Knapsack instance generator. `http://www.diku.dk/˜pisinger/generator.c`, 1994.

[21] B. Shumeet. A massively distributed parallel genetic algorithm. Technical Report CMU-CS-92-196, Carnegie Mellon University, School of Computer Science, October 1992.

[22] I. Sommerville. *Software Engineering*. Addison-Wesley Publishers Limited, 2000.

[23] D. Stracuzzi. Some methods for the parallelization of a genetic algorithm. `http://www.cs.umass.edu/˜stracudj/genetic/dga.html`, May 1998.

[24] M. Van Steen, A. Tanenbaum. *Distributed Systems Principles and Paradigms*. Prentice Hall, 2002.

[25] B. Zorman. Creation and Analysis of a JavaSpace-based Distributed Genetic Algorithm.. Technical Report CS02-18, Department of Computer Science, Allegheny College, Meadville, PA. 2002.