Annales UMCS Informatica AI 6 (2007) 23-36

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Distributed genetic algorithm implementation by means of Remote Methods Invocation technique – Java RMI

Łukasz Maciura*

The Bronisław Markiewicz State School of Higher Vocational Education in Jarosław, Czarnieckiego 16, 37-500 Jarosław, Poland

Abstract

The aim of this work is distributed genetic algorithm implementation (so called island algorithm) to accelerate the optimum searching process in space of solutions. The distributed genetic algorithm has also smaller chances to fall in local optimum. This conception depends on mutual cooperation of the clients who perform separate work of genetic algorithms on local machines.

As a tool for implementation of distributed genetic algorithm, created to produce net application Java technology was chosen. In Java technology, there is a technique of remote methods invocation – Java RMI. By means of invoking remote methods, objects between the clients and the server RMI can be sent.

To test the work of genetic algorithm, searching for maximum function of two variables which possess a lot of local maxima and can be written by means of mathematical formula was chosen.

The work of the whole system depends on existence of the server on which there are registered remote services (methods) RMI and clients, each one on a separate machine. Each of the clients has two threads, one of them accomplishes the work of local genetic algorithm whilst the other accomplishes the communication with the server. It sends to the server a new best individual which was found by the local genetic algorithm and takes the server form with the individuals, left there by other clients.

To sum up there was created an engine of distributed genetic algorithm which searches the maximum of function and after a not large modification can be used to solve every optimization problem.

1. Introduction

To accelerate the optimum searching process in space of solutions for genetic algorithm, distributed genetic algorithm (so called island algorithm [1]) was implemented. On the whole, a characteristic feature of classical genetic algorithm is that it has small chances to fall in local optimum. It is a very positive feature which distinguishes it from other heuristic algorithms, but

E-mail address: l_maciura@pwszjar.edu.pl

nothing is without a defect. Unfortunately, this algorithm has this negative feature, that searching of global optimum lasts much longer than in other heuristic algorithms. It is essential to aim at the acceleration of these algorithms. The simplest solution appears to be descent to low-levelled programming, although, this solution makes it difficult to put into practice genetic algorithm for solution of different optimization problems, besides, the speed of working increases only several times.

To accelerate repeatedly working of any algorithm it is necessary to paralle or distribute it. Paralleling depends on the fact, that application which accomplishes this algorithm has a lot of threads and each of them accomplishes the separate kind of working. In order to parallel leads to acceleration of the algorithm, this machine on which the application is run must have a lot of processors or multi-threading processors, so that each thread can be run on a separate processor or a core of processor. Distributing an algorithm consists in working which is divided using lot of machines and each of them accomplishes its separate part. These machines communicate with each other through the local net or the Internet. Most often there is also the main server on which there are common resources and which manages the work of the whole distributed system. Distribution of algorithm has this advantage in comparison to its parallelization that the number of machines in net is unlimited, however, in the multiprocessor machine the more processors are is, the more complications occur with the selection of the proper hardware to operate with any number of processors. Therefore distribution of the algorithm not its parallelization was chosen.

Nowadays, there are a lot of technologies which assist in creation of distributed systems. Some of them are independent of platform and programming language as DCOM, CORBA, others are created for specific programming language or platform as RMI mechanism in the Java technology [2,3] or Remote mechanism in the .NET platform [4]. There is also a possibility of using ordinary TCP/IP sockets but it would be work from basis in coding/decoding of objects and its packetizing through net, so the best way is to use the already checked solution. As a technology to work out the distributed system, in this work Java and its mechanism of Remote Method Invocation (RMI) were chosen. Although they say that Java, despite its improvements, is still slower than C++, in fact, the speed of programs which are working on Virtual Java Machine, systematically is approaching with its next versions to the speed of programmes created in C++. There is also a possibility, that in the future programmes in Java will be faster than those in C++. It can take place if Virtual Java Machine is realized by hardware. Besides, Java is very convenient in programming and object orientated in higher level than C++. Regarding this, for accomplishment of this work this technology was chosen.

The presented distributed genetic algorithm took its pattern from the island algorithms. They have a specific number of population processed on separated clients' machines which resemble islands. From time to time, the best individuals exchange among islands. This system differs from a classical genetic algorithm in such a way, that in this system each of the clients communicates only with server, by sending there the best of individuals and taking individuals which were left there by other clients. However, in the classical island algorithm exchange of individuals takes place between clients-neighbours which are organized in ring's topology.

2. Working of local genetic algorithm

Genetic algorithm belongs to the group of heuristic algorithms [5], which do not search whole space of solutions, but they work systematically going in some direction or directions of searching, which in a particular moment seems to be the most optimal. To the group of heuristic algorithms belong, among others. Taboo search, ant's algorithms, evolutionary algorithms. Most of these techniques were created on the basis of observation of nature and man. Evolutionary and genetic algorithms were created on the basis of transferring nature evolution methods on the computer science area. The area of working genetic algorithms is, among others, solving optimization problems.

The whole idea of this solution depends on the existence of population of specific number of individuals and each of them has one or several chromosomes which are a sequence of bits or other data representing single genes, thanks to which they can intersect with each other and be mutated. Each of the individuals presents a specific solution of the problem which is suitably coded in a chromosome. Besides a chromosome, each of the individuals has a function of the adaptation which determines, which of the individuals (solution of the optimization problem) are better and which are worse. The individuals or descendants of the individuals which have the best function of the adaptation, have the highest chance of passage to the next epoch, however, the individuals or descendants of the individuals which have a worse function of the adaptation have small or no chances depending on a method of the selection which was applied. Thanks to it every next epoch we have better and better collection of the individuals – the evolution of whole population lasts. Thanks to this strategy, separate solution is not favouring but a lot of best solutions that decrease chance of falling in local optimum. This feature of genetic algorithm presents it in favourable light in relation to other heuristic algorithms.

The local genetic algorithm that works on single client's machine presented in this work distributed system, works in the same way as a classical genetic algorithm, with such a difference that sometimes the number of individuals in

population is higher, when taking the best individuals from server coming from other clients. As a problem of genetic algorithm, necessary to test its working searching for a maximum function of two variables which possess a lot of local maxima [6] was chosen.

$$F(x,y) = 2000 - 64 \cdot \left(\sin\frac{x \cdot \pi}{16} + \sin\frac{y \cdot \pi}{16} \right) - 0.185 \cdot \left((x-64)^2 + (y-64)^2 \right).$$

For this problem it is easy to determine the function of adaptation, because it can be written as a form of mathematical formula. A single individual of created algorithm has one binary chromosome in which, there are coded two real values which make solution to the problem. We assume that these values belong to the range <0,128>. If we assum that $c_1 - c_n$ are genes of chromosome, so values x and y [6] can form the equations:

$$x = \sum_{i=1}^{n} c_i \cdot 2^{-i} \cdot 128 ,$$

$$y = \sum_{i=n+1}^{n} c_i \cdot 2^{(n-i)} \cdot 128 .$$

As a selection technique to the next epoch of individuals designed for reproduction the most popular method – roulette was applied. It depends on application of virtual roulette in which each of the individuals has its own segment proportional to the value of its function of adaptation. In practice there are ranges of real values from the range <0,1>. Then, there is a drawing of a value from this range and checking to what range it belongs. An individual which is associated with this range is admitted to the reproduction. Depending on this, if intersection follows or not, either it or its descendants come to the next epoch.

Source code for selection of a number of the individual:

```
private int SelectIndividual()
{
   double rand=Math.random();
   int id_individual=0;
   for( int i=0; i<individuals.size(); i++ )
      if( segments[i] > rand )
      {
         id_individual=i;
         break;
      }
   return id_individual;
}
```

Reproduction occurs always on individuals sorted in this way If the intersection occurs (the random value is smaller than probability of the intersection) the descendants of the parents move to the next epoch. However, if the intersection does not occur, parents move to the next epoch. If the intersection occurs, the position of the intersection is randomized and the first of the descendants receives a fragment of the chromosome from the beginning of the position of the intersection from the first parent, however, from the second parent it receives a fragment of the chromosome from the position of the intersection to the end of the chromosome.

The example of the intersection:

1st parent: 00100111 | 11010111010111111111000010100

2nd parent: 11111000 | 00101011100110011101011011 01

("|" - position of the intersection which was chosen as a result of the drawing).

As a result of the intersection, if it takes place, the following individuals arise:

1st descendant: 00100111 00101011100110011101011011 01

2nd descendant: 11111000 11010111010111111111000010100

This algorithm uses a selection of the parental pool and creates new individuals, as long as the population of the new epoch files. With every reproduction there is some probability that after carried or not carried out operation of intersection mutation occurs.

The class "Family" which realise the genetic operators:

```
import java.util.Random;

public class Family
{
   private Individual parent1;
   private Individual parent2;
   public Individual descendant1;
   public Individual descendant2;
   private static double pintersection=0.3;
   private static double pmutation=0.1;

   public Family(Individual p1, Individual p2)
   {
      parent1=p1;
      parent2=p2;
      Operators();
   }
```

```
    public static void initp(double pi, double pm)
    {
        pintersection=pi;
        pmutation=pm;
    }

    private void Operators()
    {
        if(Math.random()<pintersection)
        {
            Random rand=new Random();
            int p=rand.nextInt(34)+1;
            boolean []chromosome1=new boolean[36];
            boolean []chromosome2=new boolean[36];
            for(int i=0;i<p;i++)
            {
                chromosome1[i]=parent1.Gene(i);
                chromosome2[i]=parent2.Gene(i);
            }
            for(int i=p;i<36;i++)
            {
                chromosome1[i]=parent2.Gene(i);
                chromosome2[i]=parent1.Gene(i);
            }
            descendant1=new Individual(chromosome1);
            descendant2=new Individual(chromosome2);
        }
        else
        {
            descendant1=parent1;
            descendant2=parent2;
        }
        if(Math.random()<pmutation)
            descendant1.Mutation();
        if(Math.random()<pmutation)
            descendant2.Mutation();
    }
}
```

To recapitulate, the single genetic algorithm epoch which comes from this work and works on the local machine operates as following:

1. Work out the value of the function of adaptation for each of the individuals in the population.
2. If the best individual in this population is better than the best actual individual from the whole algorithm, then choose it as the best and set a flag which informs about this, that it has to be sent on the server.
3. Do genetic operators (intersection and mutation) as long as a new population will create from nothing.

3. The description of the distributed genetic algorithm

The system created in this work is based on mutual communication of clients – agents that accomplish the local genetic algorithm. This communication consists in an exchange of the best individuals among the agents. Each of the clients communicates with the server by means of mechanism of the Remote Method Invocation – Java RMI. By means of an invocation of the appropriate methods (functions) on the server, it can place there its best individuals as well as take those left by other agents. Mechanism of serialization of objects in Java technology allows sending in this way whole structures of objects which are placed on RAM of the computer, not only to reference to them.

4. The description of a single client

The single client is realized by means of two threads

a) The thread which accomplishes an operation of genetic algorithm.
b) The thread which accomplishes a communication with the server and an exchange of individuals.

Both threads communicate with each other by means of appropriate flags. The division to the threads is necessary not to interrupt the working of genetic algorithm during the communication with the server because it can work at this time.

5. The description of the algorithm on client's side

1. Client's thread B is logging to the server, invoking the remote method: *int login(),* and a name tag in the form of the number is assigned to it.
2. Thread B serially checks, by means of invoking on the server the remote method: *boolean permission(),* which turns a value 'true' if the expected number of logged clients to a server will be achieved. In such a case algorithm comes to point 3.
3. To establish the individuals of the population in such a way that every now and then a new best individual does not occur, thread A carries out a specific number of genetic algorithm epochs, and at the same time updates the best individual from the algorithm's start.

4. After carrying out a specific number of epochs, threads A and B start to work simultaneously (Fig. 1).

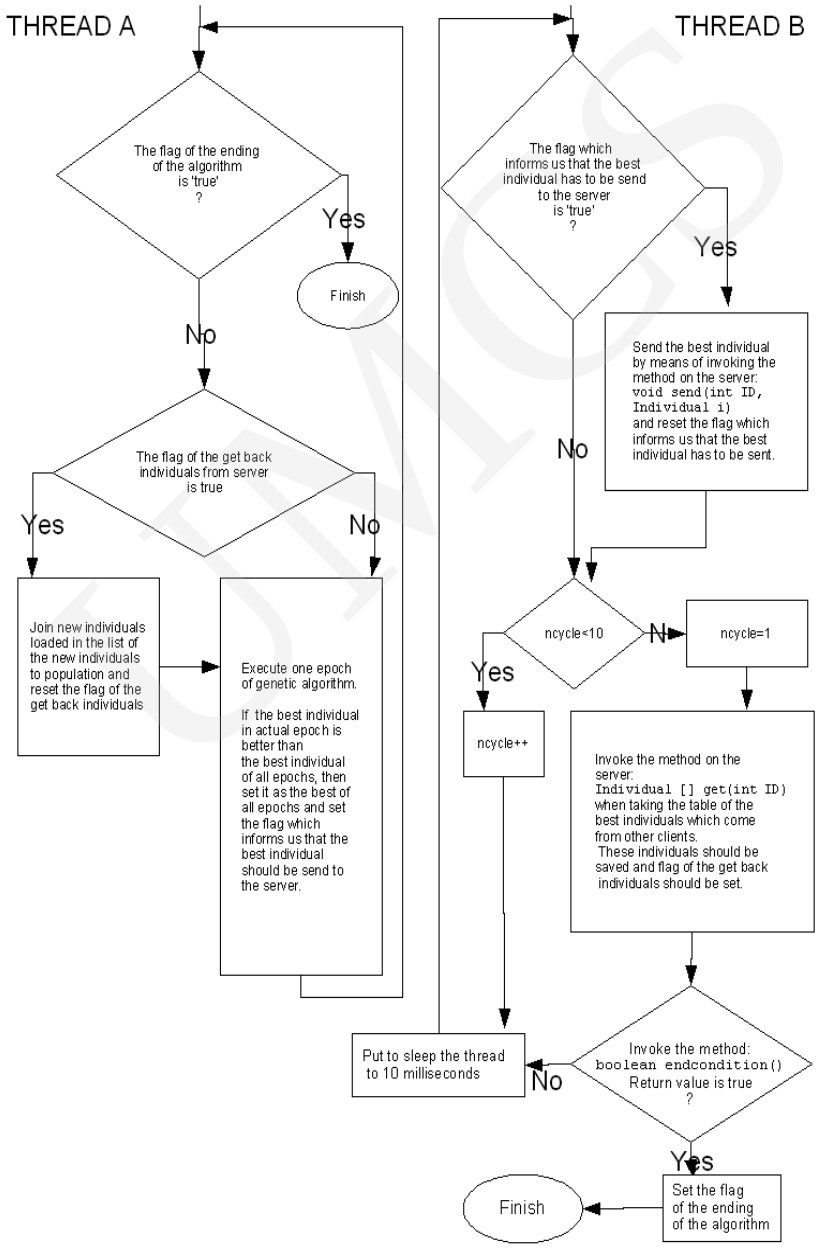

Fig. 1. The algorithm of the client

6. The description of the server

The server is in this distributed system as a relay of the best individuals among clients. It makes registered remote methods available for communication with clients and transfer of objects between them. Besides remote methods it has a table of individuals on which individuals from clients are saved. In this table each of the clients has assigned its index received at the time of logging. Besides this table there is also a matrix of value logical type, on which there is saved which of the clients load an individual which comes from another specific client. It will be needed so as a given client does not load repeatedly the same individuals from the server which could overload the server and the whole algorithm. The best global individual is also saved on the server. After the start of the server, the number of the clients which should log in, should be inserted in order to start the whole algorithm after a log in all clients.

7. The description of the remote methods

int login() – the method which is used to log in a client to the server and give to it a name tag.

boolean permission() – the method which returns the value of the logical type 'true' if a client can start its algorithm and 'false' if not. It depends if the established earlier number of the logged in clients is achieved and the flag 'start' is set.

boolean endcondition() – the method which returns the value 'true' if the condition of the end of the algorithm was fulfilled. This is established on the server and different conditions of the ending can be considered.

void send(int ID,Individual i) – the method which is used to send the best individual to the server. It is placed in a table in a position determined by ID. Additionally, there will be set suitable logical values in a matrix that specifies which of the clients loads the individuals which comes from individual clients. In the whole column there are set values 'false' because the new individual has not been loaded by anybody yet (Table 1). If necessary there is also actualized the best global individual on the server.

Table 1. The example of a content of this matrix after sending an individual by client no. 1

	Individual 0	**Individual 1**	**Individual 2**	**Individual 3**
Client 0	*true*	***false***	*false*	*true*
Client 1	*true*	***false***	*true*	*true*
Client 2	*false*	***false***	*false*	*true*
Client 3	*false*	***false***	*true*	*false*

```
public void send(int ID,Individual i)
{
   if(ID>=0 && ID<n_clients && !theend)
   {
      table_of_individuals[ID]=i;
      for(int ind=0;ind<n_clients;ind++)
         matrix_of_loading[ind][ID]=false;
      i.Print();
      if(i.Value() > threshold)
      {
         System.out.println("The end, MAX Value="+i.Value());
         theend=true;
      }
   }
}
```

Individual [] get(int ID) – the method which is used to load the table of the individuals saved by all other clients except for our own. ID is used to prevent an individual from actual client and to set the suitable value in the matrix that determines which of the individuals were loaded by specific clients. This matrix is especially needed here because it enables us to load a table only of these individuals which were not loaded by a given client (Table 2). Thanks to this, it is not possible to load the same individuals by a client. The load is possible only when an individual is on a given position.

Table 2. The example of a content of this matrix after loading an individual by client no. 1

	Individual 0	**Individual 1**	**Individual 2**	**Individual 3**
Client 0	*true*	*false*	*false*	*true*
Client 1	***true***	***true***	***true***	***true***
Client 2	*false*	*true*	*false*	*true*
Client 3	*false*	*true*	*true*	*false*

The value ‘false’ means that a given individual has not been loaded by a given client, yet, however, ‘true’ means that there are no individuals yet or a given individual was loaded to a given client. Individual 0 comes from client 0, individual 1 from client 1 etc. When a new individual is sent by client (e.g. Client 1) in the whole column ‘Individual 1’ values ‘false’ are written.

```
public Individual [] get(int ID)
{
   if(ID>=0 && ID<n_clients && !theend)
   {
      ArrayList list=new ArrayList();
      for(int i=0;i<n_clients;i++)
      {
         if(!matrix_of_loading[ID][i])
         {
            list.add(table_of_individuals[i]);
            matrix_of_loading[ID][i]=true;
         }
      }
      Individual []tab=new Individual[list.size()];
      for(int i=0;i<list.size();i++)
         tab[i]=(Individual)list.get(i);
      return tab;
   }
   else return null;
}
```

8. The algorithm on the server's side

1. Initiation of all the pools of the matrix that specify which of the clients loads the individuals which comes from individual clients to 'true', in order to block loading from the server because there has been no individuals sent by clients, yet.
2. Loading of the required number of logged in clients.
3. Waiting as long as the required number of logged in clients will be achieved.
4. Setting of the flag 'start' thanks to which the clients get to know through the 'permission' method that they can start.
5. At this moment the main programme of a server does nothing, besides the continuous checking if the condition of the ending of the algorithm is not fulfilled. If it fulfils, the best individual is introduced and the flag 'stop' is set. Remaining work is done by remote methods invoked by clients on the server's objects.

9. The system testing

To check to what extent the system increases the speed of finding the optimum in the space of solutions series of experiment, which depends on

testing the working of an algorithm on many computers, was carried out and in which the number of computers was constantly increasing. On the server, there is a threshold. After crossing this threshold the algorithm finishes its working and displays the time of working. Thanks to this, we can compare periods of algorithm calculation for different number of clients which work on separate computers. For a given number of computers, 5 tests were performed and the median of working time was calculated.

1st series of the experiment:
Settings of the algorithm:

probability of intersection	0.6
probability of mutation	0.1
number of individuals	150
threshold of algorithm end	2107.417

Table 3. 1st series of experiments

	Number of clients			
Test	**2**	**3**	**4**	**5**
1	32.5s	1.2s	0.7s	1.11s
2	2.2s	4.32s	0.7s	0.7s
3	7.05s	1.31s	0.8s	0.61s
4	5.53s	1s	2.22s	1s
5	3.11s	0.91s	0.61s	0.61s
Median	**5.53s**	**1.2s**	**0.7s**	**0.7s**

As follows (Table 3) it is noticeable that when the number of clients working on separate machines is increasing, the speed of finding of the optimum about function of adaptation higher from the given threshold is also increasing. However, it is noticeable that approximately for 4 or 5 computers the time of finding the optimum is the same. It is due to the fact that at the beginning of the algorithm working there is a big movement in web because very often a new best individual is found. It slows down working of the algorithm, because finding the optimum is short, so this delay is here very essential and it equalities time of finding the optimum for 4-5 clients. In order to see the difference for a larger number of computers we should extend the time of searching the space of the solutions. It can be caused by establishing the threshold of ending the algorithm which is adequately higher.

2nd series of experiments:
Settings of the algorithm:
probability of intersection 0.6
probability of mutation 0.1
number of individuals 150
threshold of algorithm end 2107.4173

Table 4. 2nd series of experiments

	The number of clients			
Test	**3**	**4**	**5**	**6**
1	1.92s	1s	0.52s	0.61s
2	0.72s	1.41s	0.91s	1.31s
3	19.42s	1.11s	0.7s	0.61s
4	16.41s	0.52s	2.02s	0.91s
5	25.05s	0.7s	1.31s	0.92s
Median	**16.41s**	**1s**	**0.91s**	**0.91s**

After this series of experiments (Table 4) it is noticeable that when the number of computers is increasing finding the optimum speeds up, and when the time of searching is short for a different number of clients, changes are invisible.

Conclusion

The distributed model of genetic algorithm in Java technology was implemented. It accelerates finding of the optimum in the space of solutions. The speed of searching increases along with the number of clients working on separate machines.

In the implemented example, this algorithm solves the problem of searching the maximum of the function which can be written by means of mathematical formula, but nothing stands in the way to solve any other problems by this algorithm. It is implemented by means of objected technique of Java language, so it is easy to adapt through the modification of some classes.

Acknowledgments

I would like to express my thanks to Galina Setlak, D.Sc., Associate Professor for helpful remarks.

References

[1] Schaefer R., *Basics of global genetic optimization*. UJ Kraków, (2002).
[2] Horstmann C.S., Cornell G., *Core Java 2*. Helion, Gliwice, I (2003).
[3] Horstmann C.S., Cornell G., *Core Java 2*. Helion, Gliwice, II (2005).
[4] Troelsen A., *C# language and the .NET platform*. PWN, Warszawa, (2006).
[5] Rutkowski L., *Methods and techniques of artificial intelligence*. PWN Warszawa, (2006).
[6] Cytowski J., *Genetic algorithms. Basis and application*. PLJ Warszawa, (1996).