Parallelisation of genetic algorithms for solving university timetabling problems

Bańczyk Karol
karol.banczyk@eti.pg.gda.pl

Boiński Tomasz
tomasz.boinski@eti.pg.gda.pl

Krawczyk Henryk
hkrawk@eti.pg.gda.pl

Gdańsk University of Technology
Faculty of Electronics, Telecommunication and Informatics

Abstract

Genetic algorithms play an important role in solving many optimisation problems. The paper concentrates on the design of a parallel genetic algorithm for obtaining acceptable and possibly good university timetables. Some known parallelisation techniques are introduced and the chosen implementation using MPI platform is shown. The master-slave management structure is assumed and the system scalability and the solution quality as function of the processing node number and population size are estimated.

1. Introduction

Genetic algorithms are used as means of solving optimisation problems [5][7][11] in a way similar to the evolution. A genetic algorithm starts with the generation phase (G), where random individuals i are created to constitute an initial population $P = \{i_1, ...i_n\}$. Each individual represents one of the possible solutions to the problem. The population is then modified in an evolutional process involving phases similar to the ones observed in nature, i.e.: evaluation (E), (natural) selection (S), crossing-over (C) and mutation (M). The algorithm stops if a stop condition (S) is satisfied. Algorithm 1 shows the pseudocode of a sequential genetic algorithm. The aim of phase E is to assign each individual an object reflecting its fitness. The S phase is about choosing a parent population P' from the whole P population. In C the data coming from pairs of parents are taken and crossed to create new individuals constituting a child population P''. In the M phase individuals from the child population are mutated, i.e. modified randomly, with some probability p_m. After that, some individuals from the P' population may be included into the P'' to avoid loosing good results. The child population P'' becomes then the new population P. The E,S,C,M phases are repeated until S is satisfied. The stop condition may be defined in many ways and finish the algorithm e.g. after a maximal number of iterations, reaching individuals of satisfactory quality or a lack of progress in solution evaluations in new generations. The aim of the whole process is to achieve possibly good results but the performance is also important. Like in nature, genetic algorithms work very slowly and there is often a need for their parallelisation. The ways of parallelisation are discussed in the next chapter.

The G,E,C,M phases are strongly domain dependent. Each of them is defined as a loop performing the same action a number of times (usually with different input). E.g. in the evaluation phase each individual is evaluated in another iteration.

Algorithm 1 A sequential genetic algorithm

```
1: [G] generate randomly population P = {i_1, i_2, ..., i_n}
2: while a stop condition not met [S] do
3:     [E] evaluate individuals in P
4:     [S] select parent individuals P' ⊂ P, |P'| = n'
5:     [C] cross individual pairs in P' creating P'', |P''| =
       n''
6:     [M] mutate some individuals in P''
7:     optionally substitute P'' := P'' ∪ P'
8:     P := P'' { |P''| MUST be equal to n}
9: end while
10: return one or more best individuals from P
```

There are many selection (S) algorithms possible. The three most popular schemes are: truncation, linear ranking and tournament selection [4]. In the truncation selection $n' < n$ best individuals from population P are chosen to create the parent population P'. In the linear ranking selection the individuals in P are sorted according to their evaluation, each getting a ranking number from 1 to n. The worst one gets number 1 and the best one gets n. n' individuals are chosen to P' with probabilities proportional to ranking numbers. In the tournament selection the parent population is created by repeating n' times the following routine: a pair or a larger group of individuals is chosen randomly from the population P and the best of this group goes to the parent

Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)
0-7695-2554-7/06 $20.00 © 2006 IEEE

population P'. All the three selection schemes depend on the relations between evaluations rather than on their absolute value or on the domain. This means that they may be called domain independent. Still a new function comparing two evaluations has to be defined every time new evaluation space is created.

Apart from the enumerated selection there exist other possible schemes. Some of them may depend not on the relations between the evaluations but on their absolute values and thus may be determined by some special evaluation type.

The paper deals with genetic algorithms and their utilisation in university timetabling. It gives a suitable task mapping into master-slave architecture. A proper communication strategy is assumed to avoid bottlenecks in the system. As a result, high-effective optimised solution for large timetables has been obtained. Section 2 describes the ways of parallelising genetic algorithms and gives account of the chosen approach. In section 3 the implementation of the timetable domain is shown and in section 4 some experimental results are presented.

2. Parallel genetic algorithms

The sequential genetic algorithm introduced in the previous section may be accelerated through parallelisation. In his review of recognised implementation techniques for evolutionary algorithms on parallel hardware Adamidis [2] distinguished the two main approaches to parallelisation: the standard and the decomposition one.

In the standard approach there is only one global population. As it was mentioned before, similar independent actions manipulating individuals are performed within each of the G,E,C,M phases. These actions are easily parallelised by being assigned to different processors. The selection phase may or may not be easy to be parallelised depending on the chosen selection scheme leading potentially to a parallelisation bottleneck. This approach is the most appropriate for a parallel machine with shared memory, where the communication times between the processors are very short. An example of such an approach is described in [1].

In the decomposition approach the population is distributed. Two major variants exist: the coarse-grained and the fine-grained one. In the coarse-grained version there is a number of relatively independent populations (islands) each assigned a different processor. In most cases each processor executes the same evolutional algorithm on its own population. The fact that there are many independent populations makes the global population more diverse. Occasionally, the individuals may migrate from one island to another. An example is described in [12].

In the fine-grained variant each processor is assigned a single individual. The G,E,C,M,S phases include only individuals in a bounded region called a deme. The demes overlap so the best individuals propagate gradually to all the processors.

The size of the population depends on many factors and has to be chosen through experiments. For problem domains allowing a relatively low number of possible solutions the population should not be too big as it will contain too many very similar individuals and the algorithm will work on almost the same individuals. The more complex the problem the bigger may be the population. Abramson [1] used a 100 individual population in his experiments on a school timetabling problem for 9 different input data sets. Tongchim [12] sets 500 individual populations per node in a distributed algorithm solving a different timetabling problem.

The authors of this paper have implemented a genetic algorithm falling into the decomposition coarse-grained category [3]. The approach has the advantage of possibly being used in a cluster environment thus enabling the usage of machines much cheaper than parallel computers. It also lets the global population have a better diversity allowing to find better results over time.

The created computation system uses a quasi master-slave model. Two variants have been taken into account:

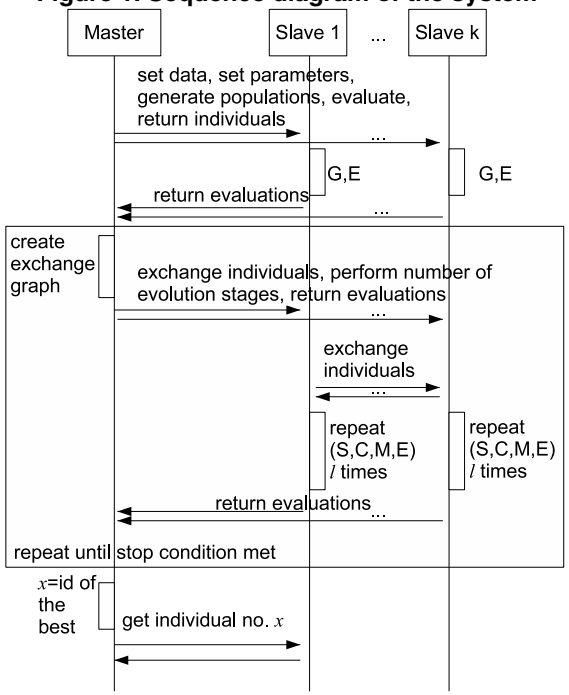
1. only the communication between the master and the slave nodes exists.

2. the communication is split into two phases, the first one is between the master and the slaves, the second is among slaves.

The second variant have been chosen as the one enabling the slave nodes exchange individuals among each other without having to involve the master node. The contrary solution could make the master node a communication bottleneck. Because of the inter slave communication the model is not a pure master-slave one.

Figure 1 shows the sequence diagram of the algorithm. The master node is the management node whose role is to send the slave nodes the instructions of what to do. The diagram does not include all the details of the system. The communication with the user is not shown. The first communication events depicted in the diagram show messages ordering the slave nodes to set the input data which describe the optimised problem, set other parameters of the algorithm, generate the initial local populations of the slave nodes, evaluate the individuals and return the evaluations to the master node. After that, the slaves perform the G and E phases on their local populations and send the evaluations to the master. Thanks to this the master knows the fitness of the populations on different slaves and is able to create the exchange graph prescribing the slaves the way of exchanging individuals among each other. The master then

Figure 1. Sequence diagram of the system



sends them orders to do the exchange according to the exchange graph, then to perform number of evolution stages and return evaluations. The slaves exchange the individuals directly with each other, do l iterations of S,C,M,E phases on their local populations and return the evaluations. The S,C,M,E phases are performed like in Algorithm 1. If the stop condition is not satisfied the process continues from the point of the creation of the exchange graph. Otherwise, the master determines the best individual and orders the slave containing it to send it to the master which in turn gives it to the user. This is the basic sequence of the algorithm but other scenarios are possible depending on the user's input.

The approach gives the master node total control over what is happening in the system. If the master did not communicate with slaves and the exchanges happened in a more decentralised way the communication overhead could be lower but the control over the whole system would be more difficult and experiments with some new exchange strategies would be impossible. The shown approach allows us to test e.g. the differences in convergence rates between various exchange strategies, using the knowledge about the global population as well as simulating the lack of the global knowledge and so creating other exchange graphs. The tests performed by the authors show that the exchange strategies using the global knowledge (like sending the best individual to all the other slave nodes) resulted in faster convergence rate than e.g. a strategy exchanging individuals between pairs of slaves. Also, the cost of the management communication may be reduced by assuming proper parameters.

The main domain independent parameters of the algorithm are: n - the size of local populations, n' - the size of local parent populations, k - the number of slave nodes, l - the number of local evolution stages between exchanges and p_m - mutation probability of a child individual. The parameter values have to be set experimentally depending on a specific problem. The larger the n the larger the local population. This leads to a better diversity of population but may waste computational power when n is too large. The larger the n' in relation to n the more individual information from previous population is used to create the new population. This results in better diversity but may also lead to slower convergence. Increasing n and l makes the local evolution stages longer. The local evolution stages should be long enough to get proper computation to communication ratio. The larger the k parameter the more power is used to compute the result but, when taken too large, it may slow down the exchange and communication phase while giving no improvement.

There are more questions to be answered. Some are related to the strategies used by the algorithm. The main domain independent strategies are: the exchange strategy determining the way in which the exchange graph is created and the local selection strategy, prescribing how the local S phase should be defined. The algorithm enables simple addition of new strategies, each of which may have its own parameters. According to Adamidis' review cited before, despite attempts to provide theoretical foundations, certain parameters still have to be set rather by intuition than analysis. These refer to the issues of which processors exchange individuals with each other, how often the exchange should take place, number of individuals exchanged at a time and the strategy used to choose the individuals to migrate.

The system working in master-slave model was primarily developed for a network environment. All the communication was encapsuled into a single communication layer, whose role is to transfer the information from master to slave and in the reversed direction. However, the layer has both a network implementation (based on MPI) and one thread version where all the nodes are indeed the objects of a single-threaded process. This fact makes it possible to compare both implementations and to check out whether and when it is useful to use a distributed environment.

3. University Timetabling problem

When implementing a specific domain using the described system the following terms have to be defined: evaluation, function comparing two evaluations, individual data, an individual, evaluation strategy and sets of the fol-

lowing strategies: the random generation strategy, mutation strategy and the crossing over strategy. In the case of the university timetabling problem some of these objects have been defined as follows:

1. evaluation $e = (h, s)$, where h = hard constraint points, s = soft constraint points. The hard constraints are the ones whose violation makes a timetable infeasible (e.g. one teacher in two classes at the same time). The soft constrains are the ones which are not hard but whose violation makes a timetable worse (e.g. the wish of having no gaps between meetings). The hard constraint points are counted as $h = \sum w_i h_i$, where w_i reflects the importance of minimising the i^{th} hard constraint type and h_i is number of violations of i^{th} hard constraint type. The soft constraints are counted as $s = \sum v_j s_j$, where v_j reflects the importance of the j^{th} soft constraint and s_j is the level of the violation of the j^{th} soft constraint type.

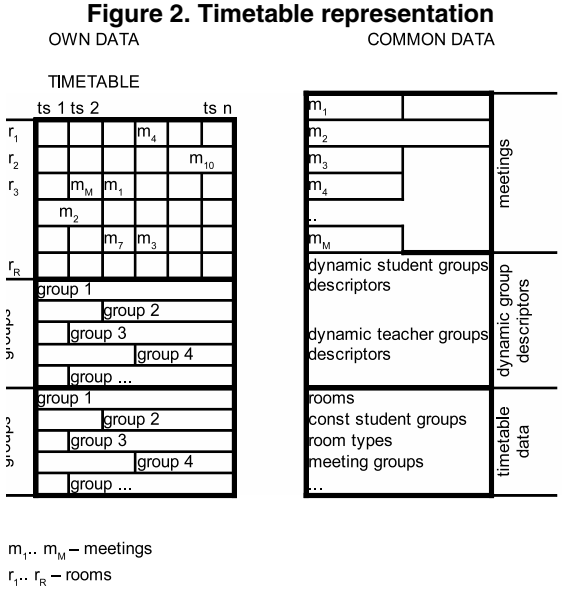
2. the evaluation comparison function compares two evaluations e_1 and e_2 comparing first their h values. If $e_1.h == e_2.h$ and $h > h_{min}$ the evaluations are defined as equal. If $h \leq h_{min}$ the comparison function returns the result of comparing the soft values s. The h_{min} defaults to 0 making the comparison function to take into account the soft value only when the hard constraints are satisfied. This is an important feature which improved the convergence over the initial version of the function where $h_{min} = \infty$ because it does not treat individuals with less soft constraint violations as better when they are not yet feasible. The $(0, 0)$ evaluation represents an optimal solution in terms of given evaluation strategy.

3. timetable data contain the information about: teachers, students, subjects, student groups, meeting groups describing meetings to take place during a set of days, rooms and their sizes, buildings, time constraints of teachers, students and subjects. The data give all the information needed to create a timetable.

4. individual i is a timetable, its representation is shown in Figure 2. A timetable consists of data common for all timetables and of private data. The common data are meetings created from meeting groups, dynamic students' and teachers' group descriptors and the timetable data given as input. The private part is the one optimised by the algorithm and contains information about assignments of meetings to rooms and timeslots as well as groups of dynamic teachers and students. In other words the algorithm is capable of splitting students and teachers into best possible groups.

This mapping of genetic algorithm terms to the timetabling domain makes it possible to express both simple weekly timetables for primary school and complex weekly timetables used in universities or examination timetables. The set of defined strategies may be easily extended by adding new or redefining the existing ones depending on the user needs.

Figure 2. Timetable representation



4. Some experimental results

The previous chapters discuss the genetic algorithm parallelisation and its utilisation for university timetabling problems. There comes the question wether it is an acceptable approach. One can answer the question through making many experiments. This paper refers to a pattern timetable with respect to 10 rooms grouped into 4 room types , 10 teachers, 20 subjects with each own meeting group and 120 students. A time span of when classes could take place has been arranged as 5 days with 6 hours of lessons each. In the preparation phase 20 meeting groups are split into 32 meetings of which 12 contain groups of dynamic students or teachers. For such input data we can create a feasible timetable i.e. the one with no conflicts. In the real university timetable problem a population consists of a number of timetables, each treated as an individual. Every individual i can introduce a number of constraint violations resulting in an evaluation e. For comparison we performed the same tests on bigger timetable consisting of 20 rooms grouped into 4 room types, 20 teachers, 40 subjects and 240 students. Initial 40 meeting groups were split by the algorithm into 64 meetings.

As mentioned before a quasi master-slave implementation was chosen. The algorithm was implemented for a dedicated cluster system consisting of 16 nodes connected with fast SCI network in Torus 2D topology. For each test a global population size was set in such a way that there were 2000 individuals per each node. There were 5000 maximum allowed iterations and additionally system stopped when there was no progress after 500 iterations. The algorithm was run using the sequential implementation of the communication layer and the parallel one on 4, 8 and 15 slave nodes. The last test was performed on 15 slave nodes instead of 16 due to hardware limitation. Simulating another slave is possible, but it would require running 2 slaves on one node thus falsifying time measurements. Test on each number of nodes were performed 100 times. The population size per node for the first pattern was discovered empirically.

Table 1. Test results (time)

	First pattern	Second pattern
Computational time (l = 1)	110 μs	210 μs
Computational time (l = 100)	120 s	223 s
Exchange time (k = 4)	1328 μs	1835μs
Exchange time (k = 8)	6016 μs	4651 μs
Master-slave communication (k = 4)	68 ms	69 ms
Master-slave communication (k = 4)	187 ms	218 ms

Some time measurements were taken. Results of those tests for first and second pattern can be seen in Table 1. It is easy to note that for such problem size communication to computation time ratio is too disadvantageous. For complex problems with more possible solutions this ratio is close to 0 as the time needed to evaluate, cross and mutate individuals increases much faster than the time needed to send an individual, which in turn depends on the size of the individual representation in memory. To further increase the ratio between computational time and communication time, the number of local evolutions should be increased. The mere increase of the number of local evolutions to 100 (thus creating communication events every 100'th iteration instead of every one ratio of computational time to communication time) increases significantly even for very small problems. For the second pattern we can observe almost linear increase in computational time both where there are no local iterations, and where there are 100 local iterations and almost no change in communication time. Independently from individual size, master-slave communication time is constant for given population size and is equal to the time needed to send number of integers equal to number of individuals. Thus it is not dependent of individual size. In this part of communication start up time pays a significant role - it is the reason why master-slave communication time increased three times when the number of nodes increased two times. Still, for bigger individuals and/or utilising more than one local evolution stages between communication acts the increasing master-slave communication time is minimal when compared to computational time. Also while computational time increases linearly with the problem, the size communication time does not change.

The tests were aimed at finding whether increasing number of nodes has an impact on successful runs (resulting in obtaining a feasible timetable) to all runs ratio (A), average time (in iterations) needed to achieve a feasible solution (B) and minimal number of iterations required to get a feasible solution (C). Results are shown in Table 2.

Table 2. Test results

No of slave nodes	A	B	C	Speedup
Sequential	0.6	488.38	66	-
4	0.67	252,52	65	1,93
8	0.7	184,47	56	2,65
15	0.71	254,25	46	1,92

It is easy to notice that even for such a small problem, running the algorithm on four slave and one master node gives a two time speedup over sequential version, where speedup is calculated as shown in Formula 1, where $T(1)$ is the time of execution on one node and $T(j)$ is the time of execution on j nodes.

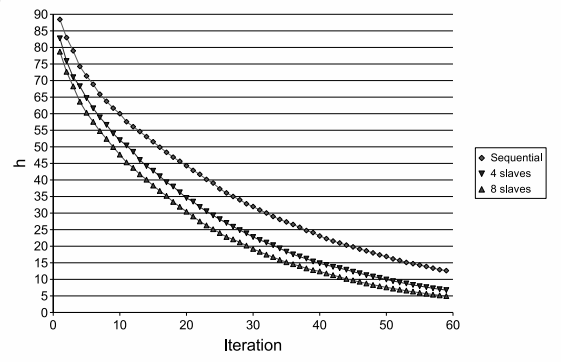
$$S = \frac{T(1)}{T(j)} \tag{1}$$

Running in parallel and thus increasing the population size without slowing down the process, enables us to achieve better results faster even for very small problems, where it is not difficult to produce all possible solutions even within the population of one node. Following the example provided, a relatively small increase of probability of obtaining feasible result was observed. This was mainly due to initial size of population. For larger problems increasing population size would have greater impact due to more possible individuals. In some situations bigger population size is required. That was the case with the second timetable patter presented earlier in this section. The algorithm on 4 slave nodes with 2000 individuals on each node made it impossible to find a feasible solution.

It is interesting to observe the situation with 15 slave nodes. Algorithm started to behave worse. Chances to receive a feasible results increased only by 1% and the average time needed to get one is longer than for 4 slave nodes.

One can notice here an overgrown population - algorithm is continously working with similar individuals thus needing longer route to get to a proper result. Although algorithm started on 15 slave nodes needed only 46 iterations to get the best result, chances on such a performance are too low to justify the costs indispensable to expand the computer systems.

Figure 3. Number of hard constraint violations h in function of iteration number



Plot in Figure 3 shows how the algorithm was in general behaving during the tests. The higher number of slave nodes proved the algorithm to have performed better within parallel environment, resulting in better results for higher number of nodes within the same iteration. This plot does not show results for 15 slave nodes as an average time needed to get to a feasible effect proved this solution not profitable.

Due to the fact, that the problem difficulty changes nonlinearly with any change of its attributes (number of students, teachers, rooms, room-type, subjects and connections between teachers, rooms, room-types, students and subjects) it is hard to give any equation giving the population size or the number of slaves needed. Following these results it can be said that it is not profitable to use more than 8 slave nodes for the presented timetable pattern. Such a boundary will exist for every timetable data of any complexity.

5. Conclusions

The most popular decomposition method producing coarse-grained parallel genetic algorithms proved to be efficient as to the means of performance and quality of solutions. Still, some parameters of such algorithms, like population size, need to be tuned through extensive tests as there are no theoretical methods known that would give their values basing on computational problem complexity.

References

[1] A. J. Abramson D. A parallel genetic algorithm for solving the school timetabling problem. 1992.

[2] P. Adamidis. *Parallel Evolutionary Algorithms: A review*. Dept. of Applied Informatics, University of Macedonia, 1998.

[3] K. Bańczyk and T. M. Boiński. *Master thesis: Distributed and parallel genetic algorithm solving timetable problem*. Gdańsk University of Technology, Faculty of Electronics, Telecommunications and Informatics, 2005.

[4] T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. *Computer Engineering and Communication Networks Labs (TIK), Swiss Federal Institute of Technology (ETH)*, TIK-Report Nr. 11, June 1995.

[5] A. Colorni, M. Dorigo, and V. Maniezzo. Metaheuristics for high-school timetabling. *Computational Optimization and Applications Journal*, 1997.

[6] D. de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19:151–162, 1985.

[7] S. Gyori, Z. Petres, and A. Varkonyi-Koczy. *Genetic Algorithms in Timetabling. A New Approach*. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2001.

[8] A. Hertz. Finding a feasible course schedule using tabu search. *Discrete Applied Mathematics*, 35(3):255–270, 1992.

[9] A. Hertz. Tabu search for large scaled timetabling problems. *European Journal of Operational Research*, 54:39–47, 1992.

[10] G. Neufeld and J. Tartar. Graph coloring conditions for the existence of solutions to the timetable problem. *Communications of the ACM*, 17(8):450–453, 1974.

[11] A. Schaerf. *A survey of automated timetabling*. Computer Science/Department of Software Technology CS-R9567, 1995.

[12] S. Tongchim. Coarse-grained parallel genetic algorithm for solving the timetable problem.

[13] A. Trimpathy. Computerized decision aid for timetabling - a case analysis. *Discrete Applied Mathematics*, (3):313–323.