Approximate Scheduling of Final Exams at Memorial University

Edward Brown, David Churchill and Manrique Mata–Montero
Memorial University of Newfoundland, Canada
brown@cs.mun.ca, manrique@cs.mun.ca

May 4, 2007

Abstract

We describe one type of scheduling problem, that of scheduling final exams at Memorial University (MUN), a comprehensive Canadian university. It is well-known that the general scheduling problem is NP-complete [1]. Over the years, this problem has attracted a lot of attention and many different approaches have been tried in finding approximate solutions [2]. Here we propose an approximate solution to the problem at MUN.We use an evolutionary programming approach [4] implemented on a commercial grid computing environment; for our experiments we used a system including 1 master and 10 slaves. The grid algorithm, not accounting for the communication complexity, achieves optimal parallelization, i.e., reduces the time complexity of its serial implementation $t(n)$, where n is the number of evolution steps, to $t(n)/NumOfSlaves$. The communication complexity on a network implementing a grid computing system may be the most important factor in the complexity of the algorithm. We alleviate the communication complexity using a feature of the software that allows for global virtual variables. These are variables that can be accessed by any of the computers on the grid and maintained by the master. On each slave we run 2 processes, since the slaves are based on dual core processors. We generate schedules better than those obtained by the current system in use. While the system is intended to be used at MUN, many universities impose very similar constrains on the scheduling of their exams. We expect the same approach to yield similar results to ours.

keywords: grid computing, combinatorial optimization, evolutionary programming, NP-completeness, approximation algorithms, scheduling.

1 Introduction

In this document we report the results of using an evolutionary programming approach to finding approximate solutions for the Scheduling of Final Exams at MUN. We used a commercial grid computing system to implement the algorithm. We ran our experiments on a system including 1 master and 10 slaves, but the number of slaves can be tailored to the available hardware. Each slave runs 2 identical processes to fully utilize the dual core technology on which the slaves are based. Apart from the communication complexity in the network, given the properties of the genetic programming methodology, we obtain optimal parallelization, i.e., without counting the communication complexity, we reduce the time from $t(n)$, on a single processor, to $t(n)/NumOfSlaves$ on 1 master and $NumOfSlaves$ slaves, where n is the number of steps of evolution. We use one of the capabilities of the grid computing system to reduce the communication between the master and the slave processes. The master is able to maintain a global virtual variable that is accessible to all the processors in the grid. In our application this feature allows for quick communication between the master and the slaves.

In subsequent sections we describe the problem, describe our evolutionary algorithm and finally present our results and open questions.

We do not describe the evolutionary programming methodology nor the terms and concepts it uses. We follow standard definitions and concepts found in the literature, for instance, in [4].

2 Description of the Scheduling of Final Exams Problem

MUN offers around 1500 courses during regular terms -Fall and Winter- and there are approximately 15000 registered students. The exam period has been traditionally set to 10 days and during each of those 10 days there are 4 time slots when students write their final exams. Some of the courses are night courses and exams for these night courses must be written during the last slot of the day -the night slot. Exams for day courses must be written during the day slots -the first three slots of each day.

There are two other fundamental restrictions, no stu-

dent must be required to write two or more exams at the same time -during the same time slot- (2-conflicts) and no students must be required to write three or more exams during a 24 hour period -in our case this means no student must write three or more exams during any period corresponding to four consecutive slots (3-conflicts).

A secondary requirement is to have as many exam papers written as early in the exam period as possible.

Solving the problem entails finding a ordered partition of at most 40 elements (the number of exams days times the time slots per day) of the set of courses fulfilling the conditions described before. This partition is called a schedule.

Clearly, a partition fulfilling the conditions stated may not exist. So, we aim at finding a schedule that minimizes the number of 2 and 3-conflicts.

3 Algorithm for Solving the Problem

We implement a grid computing evolutionary programming algorithm where each individual -phenotype- corresponds to one schedule and the genotype is a function mapping courses to time slots. So, if C represents the set of all courses and S the set of all time slots, each genotype is a function $g : C \to S$. This function induces a natural partition of C corresponding to the phenotype -schedule.

The fitness of each individual is a certain constant K plus a number of rewards minus weighted values for the number of 2-conflicts and 3-conflicts.

Notice that the fitness function reflects a slight overestimation of the number of students that have conflicts since 2 and 3-conflicts may overlap.

When there are no students with any conflicts, the fitness function reaches a value $K + r$. The constant K is an arbitrary positive integer number designed to maintain the fitness always positive and and $r \leq maximumNumberOfRewards$ is a value corresponding to the numbers of rewards obtained.

Since it is desirable to have as many exams written as early as possible, we implement this condition by rewarding schedules that during the first 2, 4, 6 and 8 days of exams enable the writing of s_1, s_2, s_3 and s_4 exams with r_1, r_2, r_3 and r_4 rewards, respectively, where $r_1+r_2+r_3+r_4 \leq maximumNumberOfRewards$. Care is taken to make sure the value of the rewards does not overwhelm the value of 2 and 3-conflicts, i.e., the rewards are a secondary goal.

As usual, our algorithm attempts to find schedules that maximize this fitness.

3.1 Computing the Fitness Function

In computing the fitness function for a particular schedule it is necessary to find out instances of rule violations among the 15000 students. This might be a time consuming task. So, we preprocess the student registration information and create a d-ary tree, where d corresponds to the number of courses, as follows.

Let the multi-set $D = \{S_n : 0 < n \leq k, k > 0\}$, where S_n is a sequence of course numbers taken by some student. To make the computation of the fitness function as efficient as possible, we set $|S_n| \leq 3$, for all n, and these sequences include all permutations of any given sequence of courses taken by a student. The value of k is then the number of all sequences of courses of lengths one, two and three -and their permutations- taken by students. Notice that a group of two or three courses is represented many times in D. This redundancy allows us to reduce the time complexity of computing the fitness function.

We recursively define a d-ary tree T, where each node includes a sequence of items $\{c, \{m\}, P\}$. The value c is a course number, the value m is the number of all the sequences in D whose prefix is given by the path in T ending with the node including c, and the third item P is a d-ary tree including all the tails of such sequences. So, given the above constraint, T has depth three.

T is large but does not change during the computation, so, it can be held in memory and it is transmitted to the slaves only once at the begining of the computation.

The tree T can be used to determine if a particular sequence of three courses -in any order- whose exams have been scheduled in any four consecutive slots has been registered by any students. The value m in the leaf node of a path of three courses specifies exactly how many student have registered that combination of courses.

In addition, the same tree T can be used to determine if a given pair of courses scheduled in a time slot -in any order- is taken by a student.

For instance, the tree T representing the registration of the courses $\{1, 2, 3, 4\}$ and $\{1, 2, 4, 5\}$ by two students is:

```
{{{1,{2},{{2,{2},{{3,{1},{}},{4,{2},{}},
                                {5,{1},{}}}},
            {3,{1},{{2,{1},{}},{4,{1},{}}}},
            {4,{2},{{2 {2},{}},{3,{1},{}},
                                {5, {1}, {}}}},
            {5,{1},{{2,{1},{}},{4,{1},{}}}}}},
  {2,{2},{{1,{2},{{3,{1},{}},{4,{2},{}},
                                {5, {1}, {}}}},
           {3,{1},{{1,{1},{}},{4,{1},{}}}},
           {4,{2}, {{1,{2},{}},{3,{1},{}},
                                {5, {1}, {}}}},
```

```
           {5,{1},{{1,{1},{}},{4,{1},{}}}}},
  {3,{1}, {{1,{1},{{2,{1},{}},{4,{1},{}}}},
           {2,{1},{{1,{1},{}},{4,{1},{}}}},
           {4,{1},{{1,{1},{}},{2,{1},{}}}}},
  {4,{2},{{1,{2},{{2,{2},{}},{3,{1},{}},
                            {5, {1}, {}}}},
           {2,{2},{{1,{2},{}},{3,{1},{}},
                            {5, {1}, {}}}},
           {3,{1},{{1,{1},{}},{2,{1},{}}}},
           {5,{1},{{1,{1},{}},{2,{1},{}}}}},
 {5,{1},{{1,{1},{{2, {1},{}},{4,{1},{}}}},
          {2,{1},{{1,{1},{}},{4,{1},{}}}},
          {4,{1},{{1,{1},{}},{2,{1},{}}}}}
}
```

Using the tree T, the serial computation of the fitness of a schedule is fairly efficient; so,we do not parallelize the computation of the fitness function.

Since we defined the fitness function as a constant K plus some reward points minus weighted values of 2 and 3-conflicts, the complexity of computing the fitness function lies in detecting such conflicts.

We give the details of detecting these violations using the tree T.

Let there be a schedule consisting of 40 time slots (ten days of four slots each), where each time slot i, where $1 \leq i \leq 40$, includes all the courses $x \in C$ such that $f[x] = i$. Since in our case a 24-hour period expands 4 slots, then to detect 3-conflicts we slide a window of length 4 from the first slot to the 37^{th}, see Figure 1. At each positioning of the window we find the number of students who are taking three courses appearing in different slots.

Suppose that the window is located at position $1 < i \leq 37$ and all slots up to slot $i+2$ have been inspected. It follows that for detecting 3-conflicts it suffices to consider 3 slots at a time, since we are detecting possible students registrations that include 3 courses exactly, one per slot.

To every pair of slots from among slots $i, i+1$ and $i+2$, we add slot $i+3$ and test for 3-conflicts. In this way we can detect all 3-conflicts *that have not been detected in previous window positions*. There are 3 pairs of slots from among slots $i, i+1$ and $i+2$, this means that we need to probe three groups of three slots, namely, all pairs from among $i, i+1$ and $i+2$ plus slot $i+3$. Too start the process we find all 3-conflicts in the slots 1, 2 and 3, and position the window in slot 1.

Given three slots, we can use the tree T to detect 3-conflicts as follows. Let the slots under test be slots A, B and C. We search the tree T and find which courses in A are courses belonging to roots of sub-d-trees in T in the root node. Then we find which immediate descendants of such roots include courses in B and identify those sub-d-trees. Finally, we test which of the immediate descendants of these latter trees include courses in C. Nodes including courses in C also include the number of students that have registered the 3-sequences found among the slots A, B and C.

To detect students required to write two or more exams in the same time slot we scan slots 1 to 40 of a schedule. For a given slot we identify in the root of T all those sub-d-trees whose roots include course numbers in the slot. Then we identify which of these have descendants that include courses belonging to the slot. The values of m in these nodes indicate the numbers of students who have registered for the corresponding 2-sequnces of courses.

So, to detect instances of 3-conflicts we slide a window from locations 1 to 37 and at each location we perform 3 tests. To detect 2-conflicts we inspect each of the 40 slots and perform one test. These tests use the tree T. So, the complexity of the fitness computation is dominated by detection of 3-conflicts, i.e., $NumberOfSlots * 3 * 3 * CostOfAnIntersection$. The cost of an intersection refers to the cost of intersecting the list of courses in a time slot with the list of courses in the roots of the appropriate subtrees.

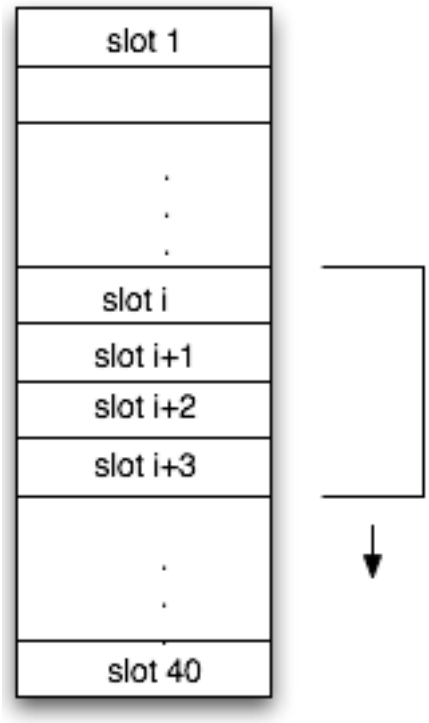

Figure 1: **Sliding a Window Down a Schedule to Detect 3-Conflicts**

3.2 Recombination, Self-reproduction and Mutation

We implemented a two point recombination scheme where for each recombination the break points are determined randomly, as described in [4].

The mutation operator changes the value of the function g defined before -genotype- in one point, i.e., for a value $x \in C$ determined at random, redefine $g[x]$ to a value y, where y is a randomly generated slot number compatible with the course number x. Recall, day courses write their exams during the first three slots of any day and night courses must write their exam in the fourth slot of any day.

We set the probabilities of self-reproduction and mutation to 8% and 27%, respectively. A probability of mutation of almost 30% goes against what many textbooks recommend. Generally, mutations are performed with low probability, around 5%. We believe that our success with a high rate of mutation is due to the way we chose to represent the individuals.

3.3 The Tasks of the Master and the Slaves

According to the evolutionary programming principles [3], an initial random population of genotypes -in our case of size 120- is used to simulate the evolutionary process of creating better adapted populations, i.e., schedules with less student conflicts -higher fitness. To create new -hopefully better adapted populations- we use the typical genetic operators, namely, recombination, self-reproduction, fitness based selection and mutation [4]. Since the process of creation of a new generation is essentially a parallel process, this is the step we choose to parallelize. We use a grid of 1 master and 20 slave processes like the one shown in the Figure 2.

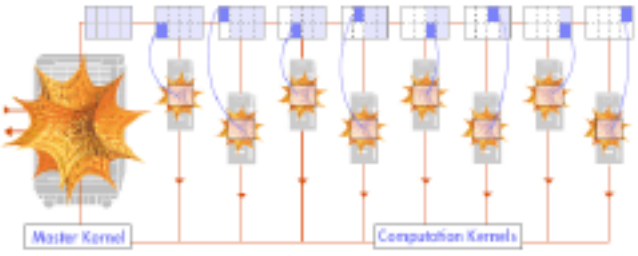

Figure 2: **Architecture of a Grid with 1 Master and 8 Slaves**

The following are the steps of the algorithm followed by the master of the grid.

- The master creates and broadcasts the tree T encompassing the student registrations to all the slaves. The master also broadcasts to the slaves all functions necessary for their task, the current population of genotypes and the fitness of their associated phenotypes.
- The master requests a number -$sizeOfThePopulation/NumOfSlaves$- of new individuals to each slave. We require this number to be even, because recombination generates two offspring.
- The master receives the $NumberOfSlaves$ segments of the new population including their fitness, genotypes and phenotypes from the slaves and assemble them into a new generation.
- The master continues the process described above until a certain number of generations has been reached or there is an individual with a fitness $K + maximumNumberOfRewards$ -an arbitrary pre-established constant plus the maximum number of rewards.

The slaves follow the next steps.

- Each slaves receives the fixed tree T, the number of new individuals it must generate, the genotypes and the phenotypes' fitness of the current population, and the functions necessary to perform its task.
- Each slave selects from the current population two genotypes -parents- using fitness based selection.
- Each slave creates two new descendants per pair of parents. New individuals corresponds to a 2-point recombination and its complementary recombination or they are identical to the parents selected -self-reproduction. In addition, these two new individuals go under probabilistic mutation, i.e., they go under a mutation based on a certain pre-established probability of mutation.
- Each slave computes the phenotypes and fitness of the two new individuals.
- Each slave repeats the process described above until it finds $sizeOfThePopulation/NumOfSlaves$ new individuals and returns them to the master.

Notice that the communication complexity is governed by the transmission of the genotypes and their fitness to the slaves. Each slave needs the fitness of all the members of the current population to probabilistically select, based on fitness, the parents of the individuals it is going to create. This information is fairly small, in our

case 120 positive integers. There are also 120 genotypes in our application, but each genotype is a list of around 1500 integers describing a schedule.

We do notice, though, that if a slave is going to create $sizeOfThePopulation/NumOfSlaves$ new individuals, it only needs $sizeOfThePopulation/NumOfSlaves$ parents, i.e., genotypes of the current generation. In our case, using 20 slave processes and with a populations of 120 individuals, each slave has to use only 6 genotypes from the current generation. So, the master does not need to transmit all the genotypes of the current generation to the slaves. Rather, it keeps a virtual global memory variable containing all the genotypes of the current generation and transmits to the slaves only the specific parents they request.

In this way we reduce the communication complexity by transmitting a total of 60 pairs of parents -genotypes- to the slaves, this, instead of transmitting 2400 genotypes per generation. The use of the global virtual variables feature greatly alleviates the communication complexity. We largely reduce the size -and hence transmission time- of the data transmitted which occurs in bursts, since the master synchronizes the creation of generations.

4 Results Obtained

Running an evolution of 5000 generations we found a schedule with fitness 14819 out of 15000. This means that there are still conflicts. There are 65 out of 13392 students with conflicts. This number of conflicts is well below the number of conflicts that the current system generates. One has to remember that there may not be a schedule with no conflicts.

The amount of time needed for 5000 generations on the dedicated grid computing system we used is of around 24 hours. This time is acceptable for the intended use of the system.

In the Figure 3 we show the fitness changes as the evolutive process takes place. We can notice that in the last 1000 evolution steps there was very little change, the conflict of 10 students was resolved. This shape of curve showing the fitness changes is common in evolutionary computations.

Even after 5000 generations, our system did not stop improving of the fitness of new populations, but the improvement was marginal and not worthwhile the amount of time required.

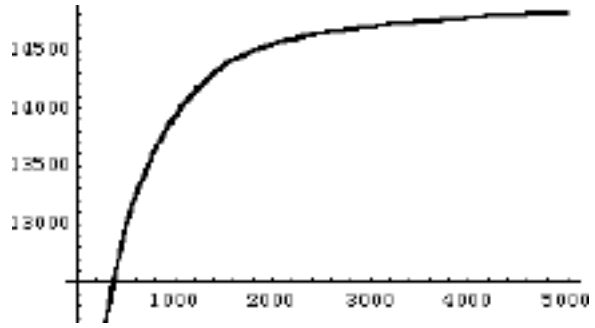

Figure 3: Fitness Changes During 5000 steps of Evolution

5 Open Questions and Future Work

The most prominent open question is whether or not we can speed up the convergence without loosing the quality of the solution. We intend to investigate this issue in the near future.

The way in which we defined the fitness function, while apparently appropriate, may not be the most useful for the practical use of the system. For instance, we may want, not only a schedule with a low number of student conflicts, but at the same time one that alleviates the task of re-scheduling the exams of the students with conflicts. In future implementations of our approach we will incorporate this extra requirement.

References

[1] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Co., San Francisco, 1979.

[2] Emma Heart and Peter Ross. Evolutionary scheduling: A review. *Genetic Programming and Evolvable Machines*, 8(6):191–220, 2005.

[3] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT, 1994.

[4] Christian Jacob. *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann Publishers, 2001.