Final Exam Scheduler

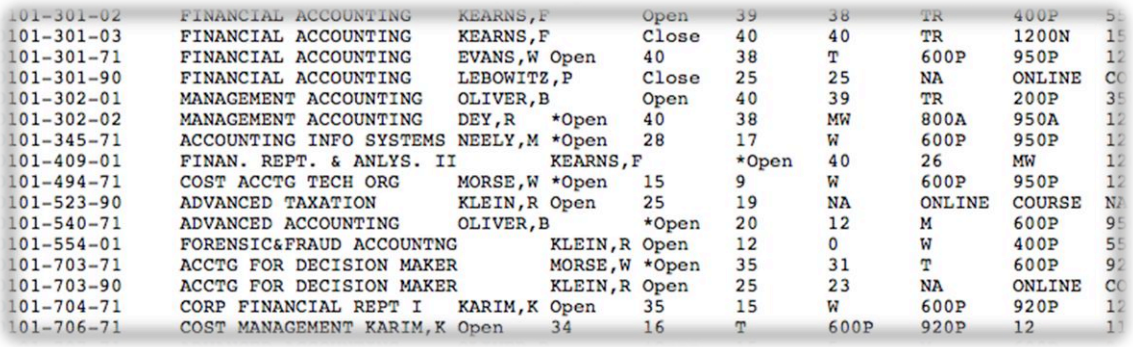
G2. Team Kyz Report

Kevin Cheek, Yandong Wang, Ziyan Zhou
Department of Computer Science
Rochester Institute of Technology
2/22/2009

Table of Contents

Computational Problem

RIT has approximately 16,000 current students and offers roughly 3,000 sections of courses each quarter. Here is an example of sections gathered from the infocenter.rit.edu website:



101-301-02	FINANCIAL ACCOUNTING	KEARNS,F		Open	39	38	TR	400P	5
101-301-03	FINANCIAL ACCOUNTING	KEARNS,F		Close	40	40	TR	1200N	15
101-301-71	FINANCIAL ACCOUNTING	EVANS,W	Open	40	38	T	600P	950P	12
101-301-90	FINANCIAL ACCOUNTING	LEBOWITZ,P		Close	25	25	NA	ONLINE	CO
101-302-01	MANAGEMENT ACCOUNTING	OLIVER,B		Open	40	39	TR	200P	35
101-302-02	MANAGEMENT ACCOUNTING	DEY,R	*Open	40	38	MW	800A	950A	12
101-345-71	ACCOUNTING INFO SYSTEMS	NEELY,M	*Open	28	17	W	600P	950P	12
101-409-01	FINAN. REPT. & ANLYS. II	KEARNS,F			*Open	40	26	MW	12
101-494-71	COST ACCTG TECH ORG	MORSE,W	*Open	15	9	W	600P	950P	12
101-523-90	ADVANCED TAXATION	KLEIN,R	Open	25	19	NA	ONLINE	COURSE	NA
101-540-71	ADVANCED ACCOUNTING	OLIVER,B		*Open	20	12	M	600P	95
101-554-01	FORENSIC&FRAUD ACCOUNTNG	KLEIN,R	Open	12	0	W	400P	55	
101-703-71	ACCTG FOR DECISION MAKER	MORSE,W	*Open	35	31	T	600P	92	
101-703-90	ACCTG FOR DECISION MAKER	KLEIN,R	Open	25	23	NA	ONLINE	CO	
101-704-71	CORP FINANCIAL REPT I	KARIM,K	Open	35	15	W	600P	920P	12
101-706-71	COST MANAGEMENT	KARIM,K	Open	34	16	T	600P	920P	12

The problem of finding a comfortable final exam schedule for all students becomes fitting 3,000 final exams into 20 time slots during the finals week. The number of possible permutations, 20^{3000}, is astronomical. Multiple algorithms for solving this problem have been studied. And we hope to show that a parallel approach will offer a significant performance benefit over a sequential approach for this problem.

The final exam scheduler is an application that computes an optimal final exam schedule for RIT students. The program uses a list of courses offered by RIT and cross-references them with the student class registration records. In order to find the best schedule, a ranking algorithm or objective function was devised to score each schedule on various criteria. Thus the resultant schedule is close to an optimal solution.

First Paper Analysis

The first research paper we studied was *Stochastic Search Algorithms for Exam Scheduling* written by Mansour and Timany published in International Journal of Computational Intelligence Research in 2007. This paper was useful in our research because it talked about some common algorithms existed in the field for the same problem that we have been trying to solve.

The paper first talked about how to model the exam-scheduling problem as a modified weighted graph-coloring problem. Their model used vertices to represent exams to be scheduled where the weight on each vertex is the number of students taking that exam. There is an edge joining two vertices if and only if there are students who are taking both exams. The weight on the edge represents how many students are taking both exams. A maximum number of colors determined by available exam slots are used to color the graph. This problem is a modification to the classical graph-coloring problem since its objective is not to minimize the number of colors used. A predefined maximum number of colors are used instead. However, both classical

and modified models aim to prevent adjacent vertices from having the same color, which in turns prevents a student from taking two exams at the same time.

The paper also provided an objective function (OF) that calculates a rank for a given schedule based on factors like number of simultaneous exams, number of consecutive exams and so on. The goal is to minimize OF, reducing the number of conflicts in the schedule for students.

The first algorithm the paper talked about was the simulated annealing algorithm (SA). SA is based on the idea from physics and is analogous to the physical annealing of solid. To coerce some material into a low-energy state, it is first heated and then cooled very slowly, allowing it to come to thermal equilibrium at each temperature. SA simulates the natural phenomenon by a search process in the solution space optimizing some cost function. In this case, the system energy is given by OF. The pseudo code for the simulated annealing algorithm is as follow:

```
Initial configuration = random chosen schedule;
Determine initial temperature T(0);
Determine freezing temperature Tf;

while (T(i) > Tf and not converged) do
       repeat (# of slots) * (# of exams) times
              generate_function();
       save_best_so_far(); // save smallest OF value
       T(i+1) = phi * T(i);
endwhile;

generate_function()
       perturb(); (randomly reassign one exam)
       if(ΔOF <= 0) then
              accept();
       else if(ramdon() < e^(-ΔOF/T(i))) then
              accept(); // accept with probability
       else
              reject();
endfunction
```

The next algorithm that the paper talked about is the genetic algorithm (GA). GA is based on the idea of natural selection. Natural selection ensures that through reproduction better individuals emerges and only the fittest survives. GA uses the idea to find the best solution in an enormous problem space. In this case, individuals are candidate exam schedules. DNAs of individuals are encoded using an array where each location corresponds to an exam and its value corresponds to a time slot. Crossover is done by randomly cloning two parents into two children, randomly selecting location k and l, and swapping all genes from location k to l between those two children. Mutation is done by randomly selecting genes in children and randomly reassigning them, in this case, changing the time slot for the selected exams. The least-fit individual is replaced with the best-so-far individual if the latter is better than the current-fittest. Convergence is detected when the best-so-far candidate solution does not change its OF value for more than 20 generations. The author also suggested hill-climbing procedure to be applied to all offspring to speed up the convergence as they claimed. Hill-climbing procedure is simply

for every gene in every offspring, randomly reassign it, and only save the change if it results in a better OF value. The pseudo code for this algorithm is as follow:

```
Random generation of initial population, size POP;
Evaluate fitness of individuals;
repeat until converge
       rank individuals and allocate reproduction trials;
       for i=1 to POP step 2
              randomly select two parents from reproduction trials;
              apply crossover and mutation;
       endfor
       apply hill-climbing to offspring;
       evaluate fitness of offspring;
       save_best_so_far()
```

This paper was used heavily in the development of our project. Our simulated annealing algorithm and genetic algorithm were both modeled after the algorithms explained in this paper. We also use some of parameters directly presented in the authors' findings.

Second Paper Analysis

The second research paper we used for this paper was taxonomy of parallel genetic algorithms. This paper provided a description of various forms of parallel genetic algorithms from simple ones such as a parallel genetic with static subpopulations to more complex ones that incorporated varying sizes of populations with and without migration. Since we have never actually implemented genetic algorithm before, this paper was extremely important to us during the initial phases. Later on, this paper also gave us ideas to parallelize a standard genetic algorithm.

The paper gave us a good general idea of how parallel algorithms worked. For the sequential genetic algorithm pseudo code provided by this paper and the Stochastic Search paper helped for forming the basis of our program. The tricky thing about genetic algorithms is that they are highly dependent upon the parameters chosen. The Stochastic search paper provided the initial parameters for the Parallel genetic algorithm.

The next problem was trying to figure out how to parallelize our sequential program. The taxonomy paper provided an overview of a number of different approaches we could take with the genetic algorithm; this led us to choose a method that utilized static subpopulations with a migration feature. We chose this method for three reasons: First, It seemed well suited to the architecture of the cluster we were running on. Secondly, it was simple to implement in the time constraints for the course. And lastly, the migration feature provided enough genetic diversity to prevent premature convergence.

As a result of choosing the parallel genetic algorithm, we were able to find an additional paper called *Parallel Genetic Algorithms with Migration for the Hybrid Flow Shop Scheduling Problem*. This paper provided empirical evidence for parameters that we could use in the parallel genetic

algorithm. Through a short investigation we decided to use parameters from both the stochastic search and this second paper to ensure benchmark's accuracy.

Third Paper Analysis

In this paper, the authors wanted to solve a challenging problem that modern parallel programs have to face. That is how to synchronize concurrent access to shard memory by multiple threads. Traditional lock-based synchronization has two crucial pitfalls which decrease the efficiency of the parallel computation. Simplistic coarse-grained locking does not scale well. With the increased number of CPU's, the speedup or the size up does not increase as much, hence the decrease in efficiency. On the other hand, sophisticated fine-grained locking may introduce deadlocks and data races.

A new way to solve this dilemma situation proposed by this paper is transactional memory. Transaction model has been used by the databases for decades. This concept transfers the synchronization operation from manual level to the system level. A memory transaction is a sequence of atom memory operations that either executes completely or has no effect to the shared memory. And it runs in isolation meaning it executes as if it is the only operation running on that memory. It makes the synchronization action only to synchronize one specific memory address instead of a whole code block in which a lot of other memory operations may exist. Transactions provide failure atomicity, which frees the programmers from having to make sure exception handlers properly restore invariants before manual releasing locks.

There are a number of ways to implement transactional memory. Eager versioning writes to the memory immediately after new data arrival while the old version is buffered in an undo log. Lazy versioning stores all new data versions in a write buffer until the transaction completes. Pessimistic conflict detection checks for conflict progressively as transactions read and write data. On the other hand, optimistic conflict detection assumes conflicts are rare and postpones all checks until the end of each transaction.

There are a few reasons as why we failed to implement the transactional memory in our research. First of all, transactional memory is still a concept that has yet to be implemented in the real world. It needs support from hardware and compiler, which cannot be done in a reasonable time frame. Furthermore, our research has taken a cluster direction instead of SMP since the early stage. Synchronization on memory I/O is not the key concerns for cluster parallel programs

Software Design

Three different approaches have been taken using different algorithms: permutation, simulated annealing and genetic algorithms. A sequential version and a cluster version have been implemented for each algorithm. FESPermutationSeq and FESPermutationClu use brute-force to navigate the entire problem space. FESSASeq and FESSAClu use SA to selectively search for an optimal solution. FESGeneticSeq and FESGeneticClu use genetic algorithm to find the best solution. Since three approaches use a similar interface and share some common classes and utilities, this section is divided into four subsections to talk about the common components and each algorithms separately.

Common components

The common component package contains numerous classes that are used by all three of the final exam schedulers.

Objects

Schedule This class stores an individual schedule. A schedule consists of the schedules rank and a mapping of what classes go into what slot. It also contains a reduction operation called REDUCE_OP for reducing schedules. This Op will look at two schedules then return the one with the lower rank for reduction operations in other classes.

Section The section class is a course with a professor, a time, the current number of students, etc… It also has a collection of student objects for students who is taking that section.

Student A student consists of a collection of sections that the student is taking, a name, and a student id. Each student based on their current courses will also provide a vote on a schedule to help determine a schedules overall rank.

Utilities

Resources The resources class provides numerous functions that are important to the overall function of our applications. The resources first and foremost provides methods for storing and retrieving student and section information from files or a mysql database. Without this ability we would not have an easy method if loading data into our application. The second function this class provides is that once all of the information is read it, it provides common accessor methods so that the information can be retrieved easily.

Random The random class has four separate methods for generating different types of random data. The first three of these generate doubles and integers with different parameters using the java.util.random class. The idea is to be able to control all random data generated throughout the program from a centralized location. The last method called generateSchedule() will create a completely random schedule. It does this by looking at the number of slots and the number of sections and randomly assigned each section to a slot then returning a schedule. This is an

extremely important function in the simulated annealing and genetic programs because of the need to initially create random pools of schedules.

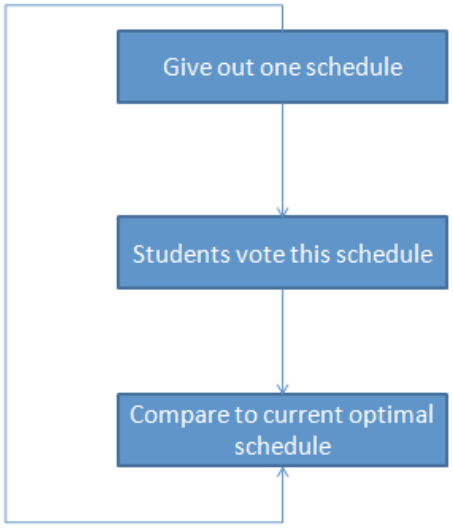
Ranker The ranker works in two distinct phases. The first thing that it does is it will examine the schedule a schedule and it will determine if the distribution of sections is relatively even with a weight variant. This removes schedules, which have a disproportionate number of final exams at one time rather than another and therefore will not be feasible. If a schedule has passed the first phase of evaluation then it will rank the schedule based on how the schedule works for each student. This is done by getting the vote from each student object in the student body then adding them to the rank. A lower rank results in better schedule.

Permutation Algorithm

The idea behind the permutation algorithm is simply iterating through the entire problem space, rank each possible schedule and find the one with the best rank.

Permutation Sequential Design

In the sequential version of the permutation algorithm, a Generator is used to iterate through all possible schedules. A Ranker from the common components ranks each schedule generated and only the best-so-far schedule is kept throughout the process.



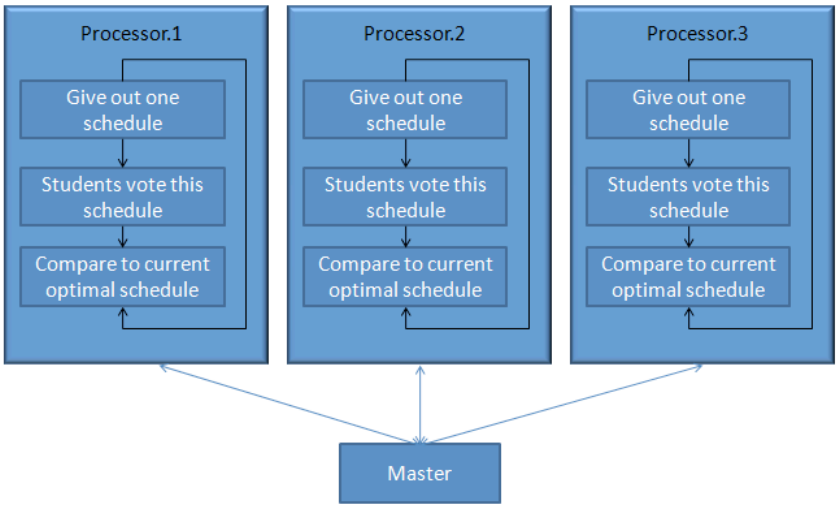
Permutation Parallel Design

The Ranker utility provided by the common package spends different amount of time on ranking the schedule based on how bad the schedule is. Obviously bad schedules are thrown out immediately to save computation time. This creates a load-balancing problem for parallel version of the permutation algorithm because nodes may finish their portion of work at

different speed. Hence, load balancing is needed. This is achieved by a master-worker approach where master assigns a chunk of work at a time and workers continuously work on assignments until all assignments are done. While all nodes run an isolated worker process, Node 0 runs both the worker and master thread.

When the program first starts up, the master sends all workers a range of schedule to be ranked. Afterwards, it blocks until a work done message is received from one of the workers. The master then sends the next range of work to that worker to keep it busy. If all ranges have been assigned, master will tell the worker to reduce its result to node 0 and exit. If all workers have terminated at that point, master will exit as well.

On the other hand, workers first wait for a range assignment message from the master. Once the range is received, the worker configures its local permutation Generator to generate only schedules within the given range. It then ranks all schedule in the given range and keeps only the best-so-far schedule. After the assigned work is done, worker sends a work done message to the master and asks for new range of schedule to work on. If there are no more schedules to be ranked, workers reduce to node 0 so that the best schedule across all nodes can be saved to a file.



Simulated Annealing Algorithm

Since the sequential and parallel designs of SA are similar, the SimulatedAnnealing class in package sa was created to contain the main body of the SA algorithm. The constructor for the class sets up the initial temperature and the initial configuration for the algorithm by calling Random.generateSchedule() to randomly generate a schedule. Method getPerturbIteration() returns the total number of iterations the algorithm should run for each temperature. Method isDone() returns true only when the freezing temperature is reached. Method getTemperature() returns the current temperature. Method runIteration(int n) runs the perturb iteration n times

for the current temperature. For details of the perturb operation, please refer to the provided pseudo code in the first paper analysis.

SA Sequential Design

The sequential version of the simulating annealing algorithm begins by loading sections, students and section enrollments information from three separate files. This is done by calling the Resources.loadFromFile() provided in the common package. Then, it initialize the SimulatedAnnealing class and run perturb iterations for each temperature until the freezing temperature is reached:

```
while(!sa.isDone())
	sa.runIteration(sa.getPerturbIteration());
```

After the freezing temperature is reached, the program saves the best schedule into a file provided from the command line arguments and prints out the timing and statistics on standard output.

SA Parallel Design

The parallel version of the simulating annealing algorithm runs on cluster nodes. The cluster program also loads data from files first. Then, the simulated annealing algorithm is initialized and run until freezing temperature is reached. For each temperature, the perturb iterations are run in parallel across multiple nodes. The number of perturb iterations to run on each node at each temperature is determined by taking the ceiling of sa.getPerturbInteration() / world.size(). After running the all iterations for the current temperature, the best-so-far solutions on all nodes are reduced into a single best-so-far solution using the Schedule.REDUCE_OP and redistributed to all nodes.

So, basically a barrier is set at the end of each temperature to synchronize all nodes. And this is done through the world.allReduce() method:

```
while(!sa.isDone()) {
	sa.runIteration(interations);

	// reduction
	buf.item = sa.getBest();
	world.allReduce(0, buf, Schedule.REDUCE_OP);
	sa.setBest(buf.item);
}
```

Once the freezing temperature is reached, node 0 saves the best schedule into a file provided from the command line arguments and prints out the timing and statistics on standard output.

Genetic Algorithm

Genetic Sequential design

The sequential genetic algorithm has two main classes. There is the ParallelGeneticSeq class found in the default package and there is a population class found in the genetic package. The ParallelGeneticSeq class is the main class and controls the population. The first thing that it does is load the sections and students from a file calling the resources.loadFromFile() function. If the data was contained in a mysql database rather than a file, this call can be replaced with the appropriate call in the resources class. The next thing that it does is creating an instance of the population class. This will create a single population to perform the genetic algorithm with. TheParallelGeneticSeq will then call the nextGeneration() function on the population which will increment the genetic algorithm by a single generation.

The population class performs a variety of activities and does the real work for the genetic algorithm. When the class is instantiated it starts by creating a random pool of individuals to match the population size. When nextGeneration() is call the meat of the genetic algorithm is performed. The first thing that happens is that a pool of individuals that will breed needs to be created. An array called wheel, is allocated with the size that matches the population size * the crossover rate. This determines the maximum number of individuals that will cross over. Once this is created the next step is to determine which individuals will breed. The first thing is that happens we take the want the top ranked individuals to breed so that percentage as defined by the TOPBREEDINGRATE. After top individuals are added to the breeding pool then added to the pool based on a probably generated by looking at the current rank and the top rank. This means that the closer the rank is to the top rank the more likely it is to be added to the breeding pool. After all of the individuals are added to the pool. Random individuals are selected from the pool and a cross over and mutations are performed on them. The crossover works by taking duplicating both of the parents then taking a random section of generates of random size and swapping them from one generated child to the other generated child. After this is done the child may be mutated based on the mutation rate set. The mutation simply takes a section and randomly places it in a new location at the end of this process. If the children created are better than a child currently in the population the child with the lower rank is replaced with a child generated, otherwise the generated child is discarded.

The last thing the nextGeneration() function will do is remove the lowest ranked individuals and replace them with new randomly created individuals. This is done because in order to benchmark the parallel and sequential algorithms effectively they both must share the same algorithm. Migration cannot be performed on a single population so replacing individuals with random ones is the simplest method to simulate the effects of migration without additional logic that would mess up the benchmarks.

Genetic Parallel Design

The parallel genetic algorithm works in exactly the same manor as the sequential except for minor changes in the main class. The first thing that is different is that each node on the cluster

gets an individual population. The main class will then iterate through the list of populations and run them a fixed number of generations, this number happens to be the same as the sequential iterations divided by the number of nodes because the goal of our benchmarks is to determine how many generations can be evaluated in a given time, not necessarily finding the best solution (Although often times the solution is very good). However, the program can be easily modified to find the best solution by checking for convergences then ending after a set number of populations have converged. It is simply for benchmarking reasons that the program operates in the manor it currently does. After the number of generations has completed, then a reduction operation is performed on all of the nodes and the best rank and schedule are printed out.

Results

The Stochastic search paper ran their algorithms with various problem sizes and listed what they were. This allowed us to create a problem of the same size and attempt to compare them. Given that we do not have the exact data used by the paper this is our best effort to compare the results directly. We randomly created a set data with 336 sections, 2456 students and 9550 enrollments to simulate the same data set the author used. Then, we ran the simulated annealing algorithm and the genetic algorithm to compare our results against the authors':

	SE	CE	ME	Rank
Our SA	0	24	351	2594.66
Their SA	3	336	572	N/A
Our GA	0	19	275	2283.26
Their GA	4	308	537	N/A

Table: comparison of the results of our SA and GA implementation against those in the paper

The criteria we used are number of simultaneous exams or conflicts (SE), number of consecutive exams (CE), and number of multiple exams or the more-than-two-per-day exams (ME). As shown in the comparison, our results have less SE, CE and ME than theirs. This showed that our algorithm was working effectively and the generated output was good. And in general, GA achieves better results than SA.

Some sample outputs from the SA and GA problems can be found in the output directory of the source package.

Performance Metrics

Simulated Annealing Algorithm Speedup Metrics

In measuring the performance of SA, we run the algorithm using 4 different data sets with 5 sections 100 students, 10 sections 200 students, 20 sections 600 students, and 50 sections 1500 students.

N	K	T	Spdup	Effic	EDSF	Devi
5	seq	11276				1%
5	1	11354	0.993	0.993		2%
5	2	6869	1.642	0.821	0.21	3%
5	3	5613	2.009	0.67	0.242	2%
5	4	4674	2.412	0.603	0.216	0%
5	8	3436	3.282	0.41	0.203	7%
5	16	2991	3.77	0.236	0.214	1%

Table SA.1: Performance measured with 5 courses, and 100 students.

N	K	T	Spdup	Effic	EDSF	Devi
10	seq	45988				6%
10	1	45807	1.004	1.004		0%
10	2	24623	1.868	0.934	0.075	1%
10	3	17538	2.622	0.874	0.074	0%
10	4	13816	3.329	0.832	0.069	1%
10	8	8372	5.493	0.687	0.066	1%
10	16	5729	8.027	0.502	0.067	3%

Table SA.2: Performance measured with 10 courses, and 200 students.

N	K	T	Spdup	Effic	EDSF	Devi
20	seq	293886				1%
20	1	291747	1.007	1.007		3%
20	2	146783	2.002	1.001	0.006	2%
20	3	101138	2.906	0.969	0.02	2%
20	4	75868	3.874	0.968	0.013	3%
20	8	40442	7.267	0.908	0.016	1%
20	16	22436	13.099	0.819	0.015	7%

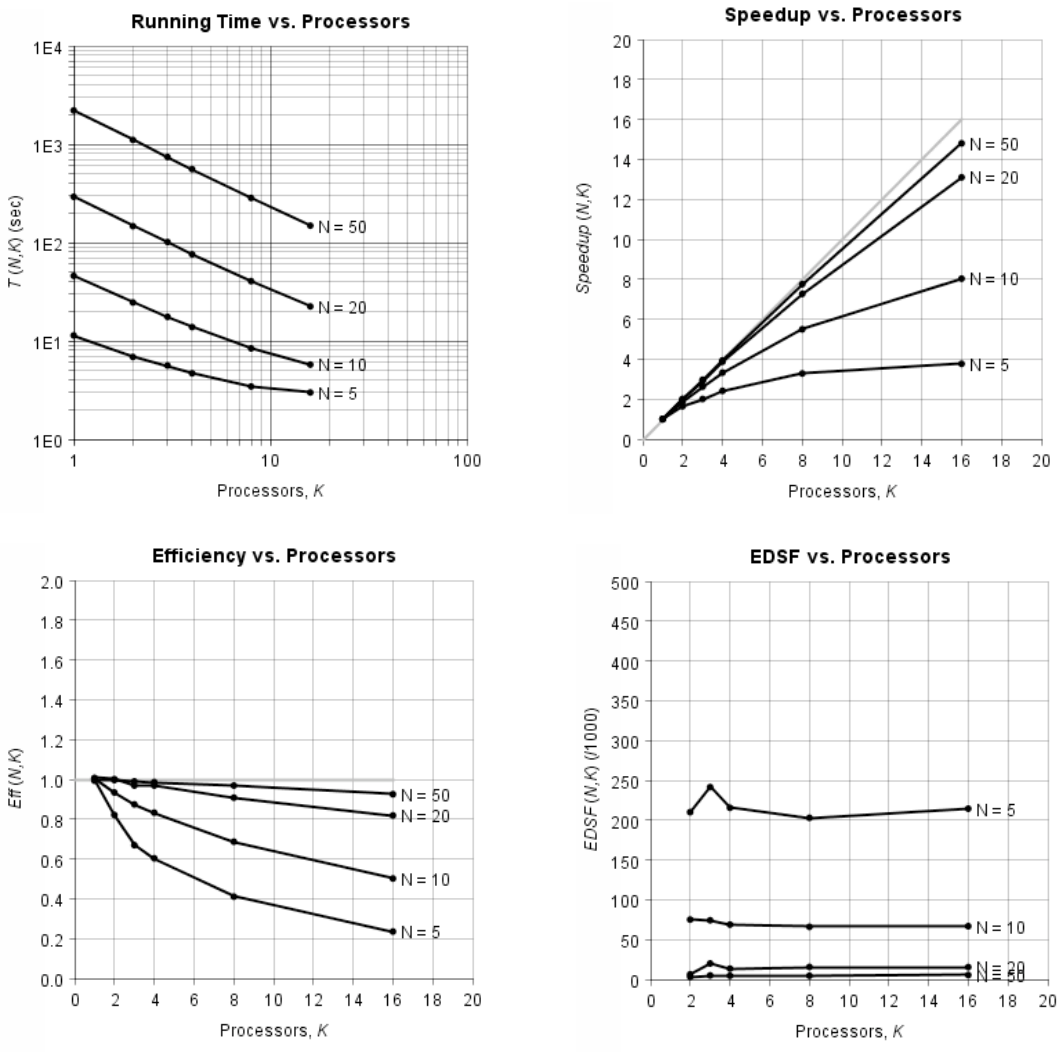
Table SA.3: Performance measured with 20 courses, and 600 students.

N	K	T	Spdup	Effic	EDSF	Devi
50	seq	2202029				2%
50	1	22026081	1			2%
50	2	11047922	0.997	0.003		2%
50	3	741593	2.969	0.99	0.005	2%
50	4	558338	3.944	0.986	0.005	2%

50	8	284154	7.749	0.969	0.005	3%
50	16	148798	14.799	0.925	0.005	4%

Table SA.4: Performance measured with 50 courses, and 1500 students.

And resulting running time, speedup, efficiency and EDFS graphs are as following:



The most obvious feature of the SA performance is that as the problem size increases, the speedup gets closer to ideal and the efficiency improves. This is because the number of total perturb per temperature is determined by the problem size. Since perturb is a random process, some schedule generated may be ranked faster than others. This creates a load balancing problem for the cluster nodes. And at the end of each temperature, all nodes need to synchronize and reduce at the barrier through calling world.allReduce(). This creates a lot of communication overhead at each temperature. But as the problem size increases, the computation time increases dramatically, and the portion of time spent on reduction becomes less significant, thus resulting in a better efficiency.

Genetic Algorithm Speedup Metrics

In measuring the performance of GA, we run the algorithm for 500 generations using 6 different data sets with 5 sections 100 students, 10 sections 200 students, 20 sections 600 students, 50 sections 1500 students, 100 sections 3000 students, 200 sections 6000 students.

N	K	T	Spdup	Effic	EDSF	Devi
5	seq	1709				12%
5	1	1702	1.004	1.004		7%
5	2	1648	1.037	0.519	0.937	14%
5	3	1872	0.913	0.304	1.15	2%
5	4	1650	1.036	0.259	0.959	7%
5	8	1843	0.927	0.116	1.095	2%
5	16	1746	0.979	0.061	1.028	4%

Table GA.1: Performance measured with 5 courses, and 100 students.

N	K	T	Spdup	Effic	EDSF	Devi
10	seq	4509				11%
10	1	4541	0.993	0.993		1%
10	2	3483	1.295	0.647	0.534	3%
10	3	3193	1.412	0.471	0.555	6%
10	4	2791	1.616	0.404	0.486	7%
10	8	2420	1.863	0.233	0.466	9%
10	16	2388	1.888	0.118	0.494	3%

Table GA.2: Performance measured with 10 courses, and 200 students.

N	K	T	Spdup	Effic	EDSF	Devi
20	seq	21182				5%
20	1	20056	1.056	1.056		7%
20	2	13204	1.604	0.802	0.317	2%
20	3	9678	2.189	0.73	0.224	9%
20	4	8221	2.577	0.644	0.213	2%
20	8	5991	3.536	0.442	0.199	1%
20	16	4736	4.473	0.28	0.185	5%

Table GA.3: Performance measured with 20 courses, and 600 students.

N	K	T	Spdup	Effic	EDSF	Devi
50	seq	132444				3%
50	1	135774	0.975	0.975		4%
50	2	74421	1.78	0.89	0.096	3%
50	3	52483	2.524	0.841	0.08	0%
50	4	41492	3.192	0.798	0.074	1%
50	8	23670	5.595	0.699	0.056	1%
50	16	14706	9.006	0.563	0.049	2%

Table SA.4: Performance measured with 50 courses, and 1500 students.

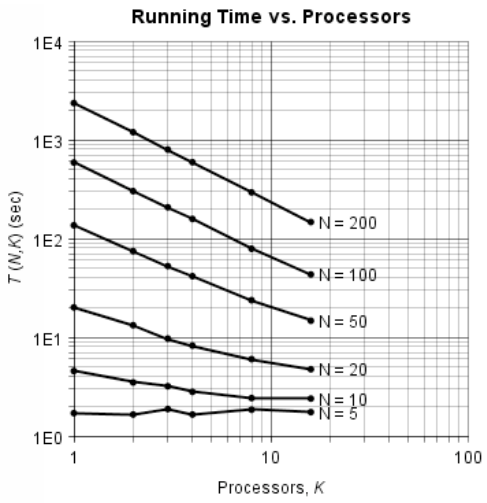
N	K	T	Spdup	Effic	EDSF	Devi
100	seq	567950				6%
100	1	586959	0.968	0.968		3%
100	2	301150	1.886	0.943	0.026	2%
100	3	206980	2.744	0.915	0.029	1%
100	4	157695	3.602	0.9	0.025	0%
100	8	78566	7.229	0.904	0.01	2%
100	16	43198	13.148	0.822	0.012	1%

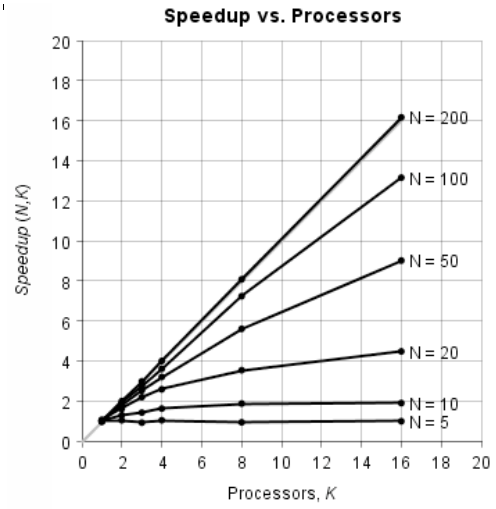
Table SA.5: Performance measured with 100 courses, and 3000 students.

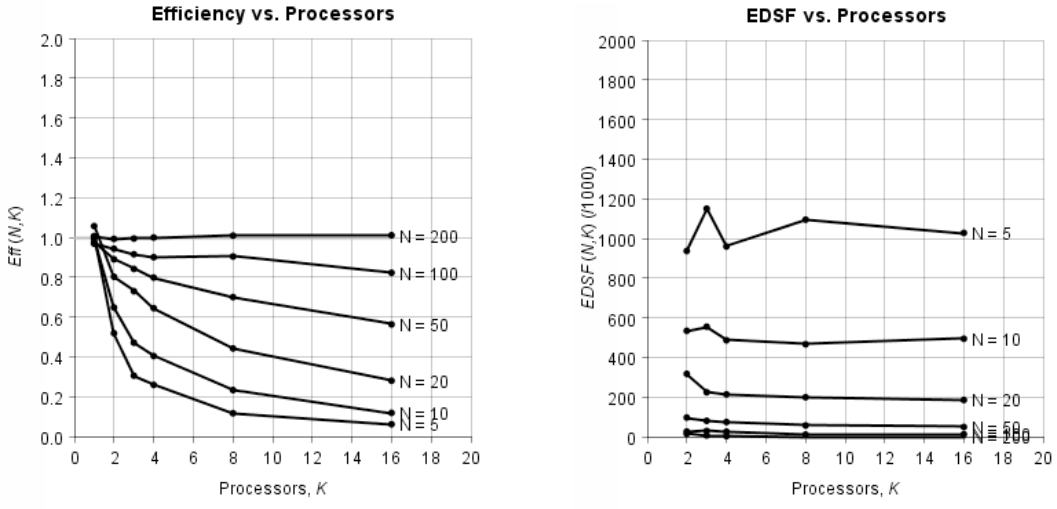
N	K	T	Spdup	Effic	EDSF	Devi
200	seq	2369339				2%
200	1	23541601	1.006			4%
200	2	11935482	0.993	0.014		3%
200	3	794773	2.981	0.994	0.006	0%
200	4	592782	3.997	0.999	0.002	1%
200	8	293621	8.069	1.009	0	4%
200	16	146560	16.166	1.01	0	3%

Table SA.6: Performance measured with 200 courses, and 6000 students.

And resulting running time, speedup, efficiency and EDFS graphs are as following:







When looking at the EDSF, Running time, efficiency, and speedup there are a number of things that can be noticed. Firstly, due to the way that the algorithm has been split up, each of the child nodes has to create its own unique population instead of sharing a single one. So in the clustered version there is a significant amount of work that has to be done on each node before they can start iterating through their generations. This resulted in a larger EDSF and lower efficiency that negatively affected the programs performance for smaller problem sizes. Once the problem size increased then creating the population pools did not take as much time relative to the rest of the computation and the efficiency and EDSF became much closer to ideal. If the problem size was small the running time also tended to be constant and not many performance gains could be made. Once the problem size increased the setup time no longer mattered as much and the running time decreased in a almost linear fashion. The speedup also exhibited similar behavior as the problem size increased the speedup per processor became almost ideal.

Future Work

There are a number of things that can be implemented or improved in the future.

First, the application needs to be tested with an increased problem complexity. Currently RIT has 2500 courses and 16,000 students. This is a significantly larger problem set that has never been tested and we would like to scale up to this real world problem size.

Secondly, the ranker can be improved in the future. The ranker currently only keeps track of Friday exams, conflicts, and consecutive exams. In the future we would like to add additional criteria for the ranker to examine.

Schedule structure can also be improved. Currently the section ID's are stored as strings, the structure should be modified to use 32bit integers in order to save space, and consequently, saving communication time when doing reduction.

The Permutation algorithm can be improved dramatically. Due to time constraints it is not as reliable as it should be.

Lastly, the genetic algorithm has the most potential to scale up to handle the real world problem sets. As a result of this, true migration should be implemented into the algorithm. Once this is done, a way to detect convergence will be needed as well as a way to detect when an ideal solution might have been found. Once populations are converging at different rates there also needs to be a way to load balance the nodes.

Lessons Learned

Throughout the process of developing final exam schedulers, there were a number of things that taught us important lessons.

Significant communication overhead between nodes were discovered when we first tried to implement the parallel version of permutation algorithm. In the first generation of the clustered permutation algorithm there was a master node and all of the schedules were transmitted to the child nodes over the network. This was extremely inefficient and later versions had a schedule generated on each node and only the ranges were transmitted between the master and child nodes.

When timing the simulated annealing algorithm, we found that some larger problem sizes that worked fine for GA would not finish in time for SA on the cluster machines. This is mainly caused by the huge number of perturb the SA has to perform for larger data sets. However, lowering the number of perturb may affect the effectiveness of the algorithm since not as many solution could be explored. The key is finding the correct balance for such situation.

In the genetic algorithm a lot of lessons were learned about the way to implement a parallel genetic algorithm. There are a large number of variations on genetic algorithms and an optimal one had to be chosen that could be implement in the time constraints that we were given. Also learned was the huge impact that the parameters had on the output of the algorithm. A lot of experimentation had to be performed to find optimal parameters that would generate good schedules. If these weren't right then the algorithm would converge prematurely on poor solutions.

Bibliography

N Mansour, M Timany. *"Stochastic Search Algorithms for Exam Scheduling"*. International Journal of Computational Intelligence Research, 2007. <http://www.ripublication.com/ijcirv3/ijcirv3n4_8.pdf>

M Nowostawski, R Poli. "*Parallel genetic algorithm taxonomy*". 1999 Third International Conference on Knowledge-Based Intelligent Information Engineering Systems, 31st Aug – 1st Sept, 1999, Adelaide, Australia. < http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=820127>

Belkadi, K., M. Gourand, and M. Benyettou. "*Parallel Genetic Algorithms With Migration for the Hybrid Flow Shop Scheduling Problem.*" Journal of Applied Mathematics and Decision Science (2006): 1-17. Springerlink. <http://www.springerlink.com/content/walc77myhp41y02l>

AR Adl-Tabatabai, C Kozyrakis, B Saha. "*Unlocking concurrency*" Queue, Volume 4 , Issue 10 (December-January 2006-2007). <http://portal.acm.org/citation.cfm?id=1189288>

Final Exam Scheduler User/Developer Manual

G2. Team Kyz

Kevin Cheek, Yandong Wang, Ziyan Zhou
Department of Computer Science
Rochester Institute of Technology
2/22/2009

Table of Contents

Compiling the Software

In order to compile the software we have included a script for compilation in the **scripts** directory of our project. Simply run "`./compile.bash`" and from the paranoia cluster and the application will compile the project.

The first few lines of the compile script include the path to JDK and other necessary jars. Change can be made to the file to better suit your developing environment.

Running the Applications

Scripted Method

The easiest method for running our application is to change to the scripts directory for our project. Within the scripts direction there is a **benchmark.bash** script. This can be executed by typing "`./benchmark.bash`" when this is done it will run the sequential and parallel versions of the permutation, simulated annealing, and genetic algorithms with from 1 to 16 nodes and on data sets of ranging from 5 to 200. If you want to selectively run tests then use the **run.bash**. This can be executed by calling:

```
./run.bash <algorithm> <K> <N> [JVM-args]
```

The **algorithm** parameter specifies which algorithm you would like to run. The values for it can either be "**Permutation**", "**SA**" for simulated annealing or "**Genetic**" for the genetic algorithm. The **K** parameter is the number of nodes that the program should run on. The value "**0**" indicates that it should run the sequential program. The **N** Parameter is the problem size that the algorithms will run with. The available data sets are in path **data/**. Currently, the valid values are 5, 10, 20, 40, 50, 100, 200, and 2500. **JVM-args** is optional and can be used to define extra parameters for the program. For example, "**-Dfes.ga.gen=100**" determines the number of generations to run in the genetic algorithm. (See detailed parameters in later sections)

All of the outputs from the run or benchmark scripts are created in the performance folder. Each type of algorithm creates its own subfolder in this directory, each corresponding with the algorithm (Permutation, SA, or Genetic) that was run. These subdirectories will then contain a timing file, an output file and generated schedules.

Manual Method

In order to run any of the applications manually data will need to be provided. Either the stock data can be used located in the data/<problem size> directory or custom data can be created. The three files located within each of the problem size subdirectories can then be used as the data files for each of the programs. Before running the application, make sure that the class path has been exported and that it references the parallel java library. Then read the section

pertaining to the application you want to run. In addition to the normal arguments for each programs there are also a number of optional parameters that can be given to the JVM with the `-D<property=<value>>` argument. The following are the common parameters that can be given to all of the algorithms.

Name	Property	Default Value	Description
Weight Variant	fes.v	1.0	Sets the weight variant
Variant Threshold	fes.vt	100.0	Determines if a schedule's slot counter varies too much
Consecutive exam weight	fes.ce	2.0	This determines how much weight is placed by a student on a consecutive exams
Multiple Exam weight	fes.me	5.0	this determines how much weight is place by a student by a student on having more than two exams in a single day.
Friday Exam weight	fes.fe	0.5	Determines how much weight is place by a student on Friday exams.
Simultaneous Exam weight	fes.se	infinity	Determines how much weight is place by a student on having multiple exams at the same time

Permutation

To run the permutation algorithm, follow the directions defined below for each of the versions. The following parameters for the JVM are common to the sequential and parallel versions and can be used by adding `-D<property=<value>>` to the java command.

Name	Property	Default Value	Description
Running time limit	fes.p.time	500	This can be a value between (0, +infinity) in milli-seconds

Permutation Sequential

To run the permutation algorithm sequentially, execute the following command:

```
java FESPermutationSeq <sections-file> <students-file> \
<relationship-file> [schedule-output-file]
```

Section File: This file is required and contains data about class sections. By default these are named “courses.txt”.

Student File: This data file containing the students taking classes. By default this file is named “students.txt”.

Relationship File: This file is required and contains information about what student is taking which course. By default this file is called “relationships.txt”.

Schedule Output File: This is an optional argument to specify the path where generated optimal schedule should be saved. If not specified, the generated schedule will be discarded. This is generally not used when the user is only interested in timing and result statistics.

Permutation Cluster

To run the permutation algorithm in parallel, execute the following command:

```
java -Dpj.np=<K> FESPermutationClu <sections-file> <students-file> \
<relationship-file> [schedule-output-file]
```

Pj.np: This property with the argument K is the number of nodes in the cluster that the program will run on.

Section File: This file is required and contains data about class sections. By default these are named "courses.txt".

Student File: This data file containing the students taking classes. By default this file is named "students.txt".

Relationship File: This file is required and contains information about what student is taking which course. By default this file is called "relationships.txt".

Schedule Output File: This is an optional argument to specify the path where generated optimal schedule should be saved. If not specified, the generated schedule will be discarded. This is generally not used when the user is only interested in timing and result statistics.

In addition to the normal arguments to be given to this application there are some other parameters that can be given to the JVM with the `D<property=<value>>` argument.

Name	Property	Default Value	Description
Scheduler trunk size	fes.p.trunk	1	This can be a value between (0, +infinity)

Simulated Annealing

To run the simulated annealing algorithm, follow the directions defined below for each of the versions. The following parameters for the JVM are common to the sequential and parallel versions and can be used by adding `D<property=<value>>` to the java command.

Name	Property	Default Value	Description
Initial Temperature	fes.sa.it	.93	This can be a value between (0, +infinity)

Freezing Temperature	fes.sa.ft	2^{30}	This can be a value between (0, +infinity)
Temperature decreasing rate	fes.sa.phi	.95	This can be a value between (0, 1)
Peturb rate	fes.sa.p	.1	This can be a value between (0, +infinity)

Simulated Annealing Sequential

To run the simulated annealing sequentially, execute the following command:

```
java FESSASeq <sections-file> <students-file> <relationship-file> \
[schedule-output-file]
```

Section File: This file is required and contains data about class sections. By default these are named "courses.txt".

Student File: This data file containing the students taking classes. By default this file is named "students.txt".

Relationship File: This file is required and contains information about what student is taking which course. By default this file is called "relationships.txt".

Schedule Output File: This is an optional argument to specify the path where generated optimal schedule should be saved. If not specified, the generated schedule will be discarded. This is generally not used when the user is only interested in timing and result statistics.

Simulated Annealing Cluster

To run the simulated annealing in parallel, execute the following command:

```
java -Dpj.np=<K> FESSAClu <sections-file> <students-file> \
<relationship-file> [schedule-output-file]
```

Pj.np: This property with the argument k is the number of nodes in the cluster that the program will run on.

Section File: This file is required and contains data about class sections. By default these are named "courses.txt".

Student File: This data file containing the students taking classes. By default this file is named "students.txt".

Relationship File: This file is required and contains information about what student is taking which course. By default this file is called "relationships.txt".

Schedule Output File: This is an optional argument to specify the path where generated optimal schedule should be saved. If not specified, the generated schedule will be discarded. This is generally not used when the user is only interested in timing and result statistics.

Genetic Algorithm

To run the Genetic algorithm, follow the directions defined below for each of the versions. The following parameters for the JVM are common to the sequential and parallel versions and can be used by adding `D<property=<value>>` to the java command.

Name	Property	Default Value	Description
Number of Generations	fes.ga.gen	500	As a result of the need to benchmark the sequential and parallel algorithms these programs do not end on a convergence condition instead they run for a fixed number of iterations.
Crossover Rate	fes.ga.cr	.75	The crossover rate is the percentage of the population that will breed each generation.
Mutation Rate	fes.ga.mr	.25	The mutation rate is the percentage of the children from the crossover that will have a mutation introduced in their genes.
Top-Breeding Rate	fes.ga.tbr	.2	The top-breeding rate is the upper percentage of the individuals that are automatically entered into the breeding pool of individuals.
Fresh blood Rate	fes.ga.fb	.1	In order to obtain accurate benchmarks the sequential and parallel algorithms needed to remain as similar as possible. As a result of this migration was not introduced into the parallel version and instead new individuals are introduced at the end of each generation to simulate the effects of a migration as close as possible. The rate at which individuals are replaced is the fresh blood rate.

To run the genetic algorithm sequentially, execute the following command:

```
java FESGeneticSeq <sections-file> <students-file> <relationship-file> \
[schedule-output-file]
```

Section File: This file is required and contains data about class sections. By default these are named “courses.txt”.

Student File: This data file containing the students taking classes. By default this file is named “students.txt”.

Relationship File: This file is required and contains information about what student is taking which course. By default this file is called “relationships.txt”.

Schedule Output File: This is an optional argument to specify the path where generated optimal schedule should be saved. If not specified, the generated schedule will be discarded. This is generally not used when the user is only interested in timing and result statistics.

Genetic Cluster

To run the genetic algorithm in parallel, execute the following command:

```
java -Dpj.np=<K> FESGeneticClu <sections-file> <students-file> \
<relationship-file> [schedule-output-file]
```

Pj.np: This property with the argument k is the number of nodes in the cluster that the program will run on.

Section File: This file is required and contains data about class sections. By default these are named "courses.txt".

Student File: This data file containing the students taking classes. By default this file is named "students.txt".

Relationship File: This file is required and contains information about what student is taking which course. By default this file is called "relationships.txt".

Schedule Output File: This is an optional argument to specify the path where generated optimal schedule should be saved. If not specified, the generated schedule will be discarded. This is generally not used when the user is only interested in timing and result statistics.

Interpreting Output

When the program has completed its execution it will print out a variety of numbers on standard output, which are in a specific format. The following is the format for the output.

```
[running time] [bestRank] [SE_Counter] [CE_Counter] [ME_Counter] [FE_Counter]
```

Running time: This is the amount of time the program took to execute.

Best Rank: This is the lowest rank that could be found.

SE Counter: The number of exam conflicts (multiple exams at the same time).

CE Counter: The number of consecutive exams.

ME Counter: The number of multiple exam problems (3 or more per day) that exist in the schedule.

FE Counter: The number of Friday exams that occur in a schedule.

If an output file was specified then the specific details of the schedule will be printed there. The output file consists of a line describing all of the counters then it will list all of the courses and their exam time.

Creating Custom data

Generate Section File

In order to create a custom problem size a section (or courses) file must be created. This was obtained from RIT's Student information system using a web crawler that uses a combination of bash and php. In order to run the script, it must be run on a unix system and have php installed.

To execute the crawler and place the output in a file run the crawler as follows:

```
./crawler.bash | php crawler.php > courses.txt
```

To execute the crawler and have the output placed in a mysql database call it in the following manner:

```
./crawler.bash | php crawler.php | php upload.php <host> <username> \
<password> <dbname>
```

Note that in order to use the mysql database option, you'll need to create a mysql database using the db.sql provided in the scripts/crawler/ directory.

After this is done the section file will be created and called courses.txt. This section file can then be used for generating data of a custom size.

Generate Data Set with Custom Problem Size

In order to generate custom random test data of a specific size you can call Generate data in the following way:

```
java FESGenerateData <sections-file> <students-file> <relationship-file>
<#-of-students> <student-load>
```

Sections file: The path to the file contains all of the courses. This was generated by our crawler script.

Students File: The path to store the student's file in.

Relationships File: the Path to store the relationship file.

Number of students: The number of students to be generated.

Student load: The average number of courses each student will be taking.