SCOOP in Practice

Research in Computer Science II

By: Mohammad Seyed Alavi
Supervised by: Dr. Sebastian Nanz
Prof. Dr. Bertrand Meyer

Student Number: 08-931-784
Table of Contents

1 Introduction .................................................................................................................. 3

2 Synchronization Problems ............................................................................................. 4
   2.1 Dining Savages ............................................................................................................. 4
   2.2 Barbershop .................................................................................................................. 4
   2.3 Hilzer’s barbershop ...................................................................................................... 5
   2.4 Roller Coaster .............................................................................................................. 6
   2.5 Search-insert-delete ...................................................................................................... 7
   2.6 Baboon crossing .......................................................................................................... 7
   2.7 Sushi bar ...................................................................................................................... 8
   2.8 Senate bus .................................................................................................................... 9
   2.9 Faneuil Hall ................................................................................................................. 10

3 Grid Computing Framework ......................................................................................... 11

4 GUI joining SCOOP ....................................................................................................... 13

5 Eiffel Thread vs. SCOOP ............................................................................................... 14

6 References ..................................................................................................................... 14
1 Introduction

Concurrent programming has traditionally been used in a number of classical application domains such as networking, high-performance computing, and operating systems. The recent availability of cheap multi-core processing hardware is expected to make concurrency also relevant for mainstream programming outside these classical domains. Therefore programming models are sought that provide abstractions to aid programmers write concurrent programs correctly and effortlessly. SCOOP - Simple Concurrent Object-Oriented Programming - is a practical framework for the development of high-quality concurrent software that carries the advantages of object technology and Design by Contract to the concurrent context. Due to its full support for advanced O-O mechanisms SCOOP claims to have the potential to address a wide range of concurrency problems.

In this project SCOOP is evaluated with respect to expressivity and performance. For the expressivity study, the goal is to investigate a broad range of concurrent applications and to develop and understanding for which types of applications SCOOP is suited best. Application domains for this study are:

- Synchronization problems
- Distributed applications
- Grid computing
- Concurrent applications with GUI representations

For reusability, all solutions are turned into a well-written program that is available for further uses also having complete documentations. Nine synchronization problems from semi-classical ones to non-classical are solved and implemented using SCOOP available as examples of SCOOP programs.

To make a new distributed problem solving framework, we have designed and implemented a Grid Computing Framework based on SCOOP programming.

Also, barbershop problem which is one of the most famous synchronization problems is implemented in both SCOOP and threaded Eiffel that makes us able to compare them from performance perspective.

The performance evaluation results will be described in section 5, preceding by GUI joining SCOOP section. In section 2, the synchronization problems are explained following by third
section which is dedicated to discussion about Grid Computing Framework and its implementation in SCOOP.

All the source codes in this study are compiled by EiffelStudio 5.7 which has SCOOP installation over it.

2 Synchronization Problems

In this section, program solutions for synchronization problems from the little book of semaphore [4] which are implemented in SCOOP are described. Problems are selected from all types of less classical, not-so-classical and not remotely classical problems. A short description of the problems and the class diagram of each solution are presented:

2.1 Dining Savages

A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Savages are the only controller processes. They check the pot before eating and if it was empty they will ask cook to fill it. Otherwise they will get the meal and eat and so on.

2.2 Barbershop

A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the
barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

Customers are the working processes. They will come to the shop, ask the shop to reduce the available seats, ask the barber to cut their hairs and then tell the shop that they are going to leave.

2.3 Hilzer’s barbershop
Our barbershop has three chairs, three barbers, and a waiting area that can accommodate four customers on a sofa and that has standing room for additional customers. Fire codes limit the total number of customers in the shop to 20. A customer will not enter the shop if it is filled to capacity with other customers. Once inside, the customer takes a seat on the sofa or stands if the sofa is filled. When a barber is free, the customer that has been on the sofa the longest is served and, if there are any standing customers, the one that has been in the shop the longest takes a seat on the sofa. When a customer’s haircut is finished, any barber can accept payment, but because there is only one cash register, payment is accepted for one customer at a time. The barbers divide their time among cutting hair, accepting payment, and sleeping in their chair waiting for a customer.
Both customers and barbers are active actors in this solution. They will interact with register which is the place for payment affairs. Each actor has a reference to the unique instance of register, shop and sofa.

2.4 Roller Coaster

Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold C passengers, where C < n. The car can go around the tracks only when it is full.

Passenger asks the car to get on and get off. The car will add or remove the passenger from its list relatively.
2.5 Search-insert-delete
Three kinds of threads share access to a singly-linked list: searchers, inserters and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

Three different types of actors try to access the shared list concurrently. All actors have a reference to the shared list instance.

2.6 Baboon crossing
There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it will break. Assuming that we can teach the baboons to use semaphores, we would like to design a synchronization scheme with the following properties;

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than 5 baboons on the rope.
A continuing stream of baboons crossing in one direction should not bar baboons going the other way indefinitely (no starvation).

Baboons are active actors who ask the rope for reserving the direction while they are going to traverse. They will mount and unmount the rope before and after traversal respectively.

2.7 Sushi bar
Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.

Bar is the critical resource that customers will compete for it. They will ask bar for entering and leaving. Each customer has a reference to the unique bar instance.
2.8 Senate bus

Riders come to a bus stop and wait for a bus. When the bus arrives, all the waiting riders invoke `boardBus`, but anyone who arrives while the bus is boarding has to wait for the next bus. The capacity of the bus is 50 people; if there are more than 50 people waiting, some will have to wait for the next bus. When all the waiting riders have boarded, the bus can invoke `depart`. If the bus arrives when there are no riders, it should depart immediately.

Making detachable list of passengers made us to make a new class of SEP_PASSENGER. This was the only way found for getting rid of compilation errors.

Passengers will come to station, ask the station for check in and then they will go to the waiting list after leaving the bus which is already waiting in the station if there is any bus waiting. Otherwise they will go directly to the waiting list. Bus will pick up passengers from waiting list and leave the station. Station is the critical resource which passengers compete for it and buses and passengers should keep it synchronized always during the scenario’s execution.
2.9 Faneuil Hall
There are three kinds of threads: immigrants, spectators, and a one judge. Immigrants must wait in line, check in, and then sit down. At some point, the judge enters the building. When the judge is in the building, no one may enter, and the immigrants may not leave. Spectators may leave. Once all immigrants check in, the judge can confirm the naturalization. After the confirmation, the immigrants pick up their certificates of U.S. Citizenship. The judge leaves at some point after the confirmation. Spectators may now enter as before. After immigrants get their certificates, they may leave.

Hall is the main critical resource which all actors from immigrants to judge and spectators compete for it. Each actor has a reference to the unique object of hall and they will ask it for their requests like enter, check in and leave.

Some technical issues that should be considered while reusing these source codes are coming below:

- If you are going to rename the project and the root class, you should also check the .ace file manually to see if the names are correctly changed in it. Then you should remove both scoop_build and EIFGENs folders and try to recompile the project using EiffelStudio 5.7 with scoop installation. Otherwise you will face an exception from SCOOP_STARTER class.
• Except the sushi bar which also has GUI representation for the output, all the others have only text outputs.
• It is recommended to remove both scoop_build and EIFGENs directory before each compilation to be sure that you will have the new running version of application after completion of compilation.

3 Grid Computing Framework

Design and implementation of many solutions for synchronization problems made me more familiar with SCOOP programming and reached me to the idea of designing a completely distributed grid computing framework that is able to get problems in a special format and solve them by the use of distributed and completely independent machines. Following class diagram shows the basic structure of grid computing frame work:

![Class Diagram](image)

There are two shared queues between MANAGER and WORKERS; one of them contains tasks and the other one has the task results. MANAGER asks the problem for new data to make a new task and put it in the task_queue. Each WORKER does a loop starts from getting one task from task_queue. It does the task, make the task result using the original instance of task and obtained result. Finally, it puts the result in the result_queue. After putting one new task in the
queue, MANAGER checks the result_queue for new result and tries to update the final calculated result.

For each problem that we intend to solve in this framework, we should implement the deferred features in WORKER, MANAGER, TASK, TASK_RESULT, DATA and PROBLEM by inheriting new classes from them.

As an example, we decided to solve the Pi number calculation problem using Monte Carlo Method[6] [7]. The following class diagram shows the inherited classes for the Pi calculation problem.

![Class diagram](image)

PI_PROBLEM generates random POINT_DATA and MANAGER makes PI_TASKS from them to put in the queue. PI_WORKER gets the task from the queue, using do_task feature it checks if the point is inside the circle or not, makes PI_TASK_RESULT with the Boolean result and puts them in the result_queue. MANAGER does the last job which is final result calculation and update according to each PI_TASK_RESULT in the queue. It will do it by using Monte Carlo formula of Pi number calculation.
Using separate objects is the ability that makes the design of framework so fast and free of considering the synchronization problems however it makes the execution time more than what was expected. During the design phase, we should not be worried about any lock passing or simultaneous access to an object when we have separate objects. However, because of many separate objects and not having process pool to assign specific processes to each separate object, program executes slowly. This problem can be solved by not letting the number of processes to exceed a certain limit.

4 GUI joining SCOOP

As it is not possible with the working version of SCOOP to make a project based on SCOOP and using EiffelVision as graphical part, we decided to implement the graphical user interface in Java and connect that to the executable file generated from Eiffel code by using C. By this way, we have a .jar file as GUI part, .dll file as the connector and .exe file which is the core execution of sushi bar solution.

The idea is based on the work of Marco Trudel who has already implemented a game with SCOOP and Java GUI.

The most important and tricky part of reusing the code and making it run, is the compilation part. It’s so easy to make the .jar file from Java source code. The next step is to make the .dll file and .lib file which is needed for the linker during Eiffel source code compilation. This job can be done by using any type of gcc compiler. The build.bat should be changed to have the new path of gcc compiler. You can also change the path of output files. After the compilation, we will have .dll file and .lib file.

To force the Eiffel compilation to use .lib file during the linkage time, we should change the linker in our C compiler. The name of original linker should be changed from “link.exe” to “link-org.exe” and substitute a new linker in the way that will be described later. In the case that you are using Microsoft Visual Studio as C compiler, this file can be found in the bin folder under VC directory in Microsoft Visual Studio installation path (e.g. C:\Program Files\Microsoft Visual Studio 9\VC\bin).

To make the new linker, you should compile the “LinkReplacement.java” file to native using JNC (http://jnc.mtsystems.ch/). Before start to compile, origLinkExe and libFile must be changed to the local paths of original version of linker (link-orig.exe) and java connector library file (JavaConnect.lib). For easier compilation to native you can use default.jnc file after adapting that according to the local paths. In step 3 of compilation wizard, the GUI and JCE should be excluded from the class library. The new linker file with name “link.exe” must be copied to the bin folder of Microsoft Visual Studio or any other C compiler that EiffelStudio uses.
After all these steps, you can put .exe file from scoop_build/EIFGENs/sushi_bar/W_code, .jar file and .dll file together and run the .exe file.

5 Eiffel Thread vs. SCOOP

Comparing SCOOP programs with EiffelThread would be the first step to make an evaluation of advantages and disadvantages of using SCOOP instead of old version of concurrent programming method in Eiffel which was EiffelThread.

We have decided to implement the barbershop synchronization problem which is one of the most famous standard problems in the concurrent programming domain in both SCOOP and EiffelThread.

There are some system delays in the code representing the time needed for customers and barber actions that should be set to the same value in both source codes before starting to profile the applications. These delays are showing the time needed for the barber to dress the hair of each customer, the time which customer needs to spend before returning again to the barbershop for next hair cut and the time which is needed to spend between customers’ constructions.

This comparison needs a complete profiler that can monitor runtime parameters of both applications during execution time. It also needs a benchmarking platform which we can test the applications without other interrupting processes running on that system.

All of these issues reached me to the decision that the evaluation and comparison of developed applications should be done in future and cannot be done as part of this project. The source codes of both developed solutions of barbershop problem are available for further uses.

6 References