Project Report: Farm Generator

Ali Tabatabaee

December 14, 2019

1 Project Description

A load-balanced farm consists of a single central process which generates new tasks and schedule them over multiple servers, several processes that work as servers and solve their given sub-problems (tasks), and also another process, called receiver, that handles the results outputted by servers.

The main targets of a farm can be listed as follows:

- A correct solution to the problem is found through the collaborative work of processes in the farm
- Workload is balanced among the servers
- Servers are kept busy as much as possible during the runtime
- The runtime for the whole problem is minimized

The objective of this project has been to write a program in Python which gets as input three functions *generate_task*(), *process_task*(), and *process_result*(), three custom data structures for *tasks*, *results*, and *analyses* (outputs of the *process_result*()) written in C, and also a configuration file and then creates a C source code for a farm consisting of N processes using MPI libraries:

- One process is the central process which calls *generate_task*() and assign the generated task to a server while managing the workload of the processes
- $N - 2$ processes act as servers and call *process_task*() to process their given tasks
- Another process works as a receiver which gets the results from servers and calls *process_result*() to further process them

2 Achievement of Milestones

In the early stages of this project, five milestones have been defined for developing the farm generator. In this section, we restate the milestones and highlight the work that has been done to achieve each of them:

1. **Developing a simple farm using MPI in C to gain more insight:** This milestone was achieved very early on the first phase of the project and the development soon stepped toward a more complicated farm.

2. **Writing a Python code to generate the code of the simple farm based on simple configuration parameters:** Due to a developmental decision and in order to decrease reimplementation, the development of the farm generator code was started at the final stage of this project and the milestone was achieved by the end of the second phase of the project. The farm generator code is written using *Cog*, which is a code generator tool written in Python. Using Cog, the Python code for reading the configuration parameters and source file and generating the related C code is embedded in the original source code. The Python code can be executed and the output of it is inserted in the middle of C source file.

3. **Adding more complexity to the farm to make the scheduling algorithms and communications more efficient**: Working toward this milestone was started at the first stage and continued until the end of the project. In this project we have developed four different scheduling algorithms for the farm and the choice of which algorithm to use is made by the user in the configuration file. These algorithms are as follows:

   - **Push-Based:** This is a traditional algorithm used by load-balancers in which center, based on information it has, pushes tasks to servers of its choice.
   - **Pull-Based:** This is another traditional load-balancing algorithm in which servers ask center for new tasks when they finish (or are about to finish) their previous tasks.
   - **Combination:** This algorithm was designed in the first phase of this project and combines Push-Based and Pull-Based algorithms for assigning tasks to servers. For each generated task, the center checks if it has task request from servers; if it does, then it assigns the task to the server that requested it or else, it selects a server based on its own information.
   - **Combination then Pull:** Based on the intuition that it is better to have only the Pull-Based approach at the final stages, in order that servers finish their jobs early at relatively the same time, this algorithm was designed in the second phase of this project. Based on the results of evaluation tests, this algorithm seems to work better than the rest for majority of cases.

   Moreover, considering data structures for task, result, and analysis and handling their communication to give more flexibility to user and adding the possibility for the receiver

to send analyses and feedbacks to the center were the other notable achievements of this project.

4. **Recognizing more configuration parameters and therefore giving the user more control on how the farm works:** During the second phase of the project, multiple configuration parameters have been detected and the user can determine them by setting the configuration file. These parameters specify:
   - The choice of scheduling algorithm between the four provided options
   - If the receiver sends analysis to the center
   - If new problem and functions are given or the default problem is used
   - The initial memory size used by processes
   - The period that the center should be updated by the receiver on finished tasks
5. **Developing more complicated real-world examples:** Complexity of different problems come from their memory requirements and runtime. Since the logic of the solution for the specific problem of the user is defined by them, we can create examples as much diverse as needed for our purpose by defining the "dummy" memory and estimated elapsed time of each task. For evaluation, we have created multiple test cases with different features using this approach.

3 Evaluation

In order to evaluate the performance of the scheduling algorithms and compare them to $SAfarm$, provided by Prof. Wagner, multiple test cases have been devised. In each test case, the number of servers is set to 5, we have 50 tasks of size $40KB$ each, the size of each result is $4KB$ and processing the results takes nearly zero time. Moreover, the initial memory size for both center and receiver is set to $400KB$ and receiver sends analyses to center. Also, slight modifications were needed to make SAfarm compatible for testing; the modified version of SAfarm is available to present. Tests are designed in a way to cover variety of problems where center is busier than servers, centers and servers are equally busy, and servers are busier than the center. The results of the tests are observable in Table 1.

Table 1: Runtime of farms given tasks of different estimated time

Farm Setting / Task Time	0.030s	0.050s	0.060s	0.080s	0.100s	0.200s
Push-Based	0.650s	0.703s	0.767s	0.977s	1.187s	2.140s
Pull-Based	0.651s	0.661s	0.740s	1.066s	1.116s	2.171s
Combination	0.650s	0.690s	0.731s	0.994s	1.186s	2.137s
Combination then Pull	0.649s	0.679s	0.720s	0.961s	1.106s	2.149s
SAfarm	0.637s	0.726s	0.767s	0.965s	1.156s	2.188s

For every test, the farm setting with the best performance is highlighted. As we can see, the scheduling algorithm *Combination then Pull* has the best performance in 3 of the 6

different tests comparing to all other farm settings, particularly in cases where servers are slightly busier than the center. Moreover, the scheduling algorithms developed in this project demonstrate good performance comparing to SAfarm. Out of the six test cases, SAfarm has the best performance only in one case and for rest of the cases, Combination then Pull works faster than SAfarm.

4 Instructions

The source codes and other deliverables of the project are available both on cyclops and the GitHub repository at `https://github.com/The-CodeBreakeR/FarmGen`.

The farm generator code is named *farmcog.c*. The configuration file for the farm should always be named *farm.config* and be kept on the same path as *farmcog.c*. The configuration parameters have been described earlier in the report. The path to the rest of files (such as the one containing the *generate_task*() function) can be provided in *farm.config*. If a configuration parameter is not determined in the configuration file, it will be set to a default value. Parameters should be determined in different lines and the name and value of each parameter should be separated by a '=' sign with spaces on both sides of it. Please refer to the sample *farm.config* file in the repository for more information.

In order to execute the farm generator, we first need to install Cog on our system. This is achieved using command:

```
$ pip install cogapp
```

Now in order to execute Cog codes and generate the farm into *generatedfarm.c* file, the following command is used:

```
$ cog -o generatedfarm.c farmcog.c
```

If we want the generator code to be deleted from the output file we use:

```
$ cog -o generatedfarm.c -d farmcog.c
```

Now we can use the following command to compile the farm code:

```
$ mpicc -o generatedfarm generatedfarm.c
```

Finally, in order to run the farm with e.g. 5 servers (7 processes), we use the following command:

```
$ mpiexec -n 7 ./generatedfarm
```

5 Resources

In addition to multiple web-resources that have been used to better learn about *load-balanced farms*, *MPI*, and *Cog*, source codes of *SAfarm*, *adlb*, and *mrlib* have been reviewed to achieve better insight on details and configurations of farms.

6 Conclusion

In this project we have developed a farm generator that creates a farm based on configurations defined by the user. The four scheduling algorithms of this farm, particularly the newly designed *Combination then Pull*, have worked well during the evaluation. Therefore, further investigating the new algorithms could be a potential research direction.

One feature missing in the farm generator was combining the center and receiver into one process. Adding this feature could be considered as a future work for this project.

The project has helped the author gain more insight on how farms and generally distributed systems work.