25/6/2016

Matrix Multiplication Through Distributed Computing using ZeroMQ

Rizwan Asif
12beerasif@seecs.edu.pk
National University of Science and Technology (NUST),
Islamabad, Pakistan.

Introduction:
This program creates a cluster which performs matrix multiplication using distributed computing. The number of clients and workers are to be specified by user. While it runs on a single server.

Problem Statement:
Design a cluster in which a task is distributed among multiple machines (resources) and executed in parallel. You are expected to develop a client server application as clients send a task to the server and server distribute that task to the resources attached to it.

Environment and Tools used:
The following tools have been used to produce this project

- Python 2.7x or Python 3
- Ubuntu 14.04
- ZeroMQ library
- JSON

Architecture:
This program utilizes a load balancing broker service to perform the task. As shown in Illustration 1.

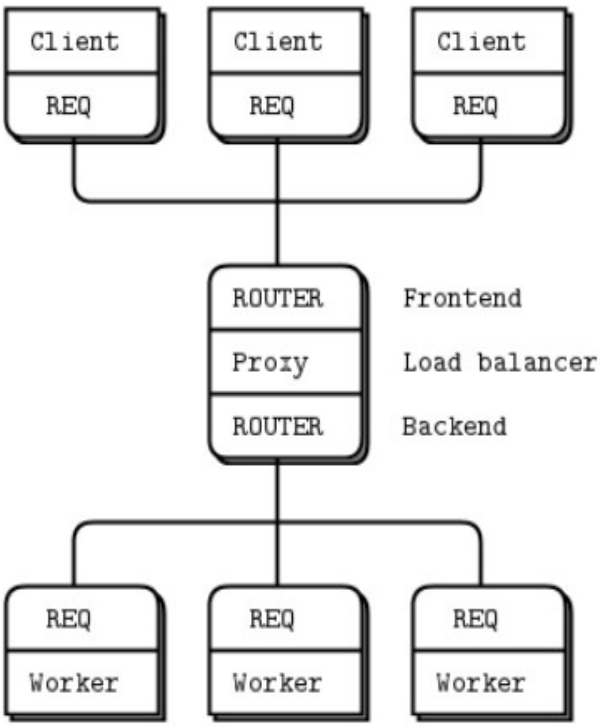

Illustration 1: load balancing broker
Courtesy of [1]

The client utilizes a REQ socket to communicate with server. The server divides the message into smaller tasks that can be distributed among workers. The workers receive messages from the server via REQ sockets as well. The communication takes place using

Multi-Part Messaging, a technique based on identification system for routing messages to different nodes. The payload however is composed of JSON objects, because of their easy encoding and decoding of python dictionaries.

The challenge faced in this architecture was the division of work to worker nodes. For this purpose a stack '*WORK_STACK* ', in the form of python dictionary, is maintained by the server. Whenever a worker is available the server pops a task from the stack and sends it to the server. The only tag associated with this task is the client address, which is attached to the multi-part message header. The server uses polling mechanism to look for available workers and requesting clients.

Matrix Multiplication:

The algorithm used for dividing the matrix multiplication problem into parallel tasks is called divide and conquer algorithm [2]. The worker was provided with one element from each matrix and they had to return their product.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\begin{aligned} &\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ &\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ &\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ &\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

(b)

Illustration 2: Divide and Conquer Algorithm
Courtesy of [3]

Two major challenges were faced in the implementation.

1. To track a client's task, after being distributed to workers.
2. To recombine individual tasks at their proper location in the resulting matrix.

The first problem arises because we cannot track which task will be processed first and intuitively one client take more time than the other. Since, there is not limit to the size of square matrices. This problem was dealt in the '*WORK_STACK*', where each individual task was assigned its relevant client id. Hence, the worker was aware of the client's address and returned that address back to the server.

The second task was dealt by creating a python dictionary '*CLIENT_DICT*', which is responsible for each clients total tasks, tasks pending, tasks received and the solution matrix. Whenever a single task for a particular client is received from a worker, then the '*RecTasks*' key in '*CLIENT_DICT*' is incremented, and the solution is updated. Moreover, in order to determine the correct position of performed task in the solution matrix, the task packet is provided with an ID. This ID provides position of this task in the final resulting matrix.

Results:

The code was tested with multiple workers and clients. The matrix multiplication was accurate, as compared using online calculators like [4]. Moreover a significant increase in speed was observed with when number of workers were increased.
With 2 clients and 1 worker, 3x3 matrices took 42 ms to multiply. Whereas with 2 clients and 4 workers, 3x3 matrices took only 9 ms to multiply.

Possible Improvements:

At this time the project is doing element by element multiplication through distributed computing. However, addition operation is still being carried out on the server. We can improve the performance further by assigning the workers to do addition operation as well.

References:

[1] Hintjens, P. (2011). Ømq-the guide. Online: http://zguide. zeromq. org/page: all, Accessed on, 23.

[2] E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing," in *IEEE Transactions on Computers*, vol. C-32, no. 6, pp. 582-585, June 1983.

[3] “Chapter 2: Parallel Programming”, http://www.darshan.ac.in/Upload/DIET/CE/SEM%208/Darshan_Sem8_180702_PP_2014.pdf, taken on June 2016.

[4] “Matrix Refresh', http://matrix.reshish.com/multiplication.php, taken on June 2016.