Distributed multiplication

Iman Hosseini

1 Two processor

In this work I am going to use the commodity computational hardware I have access to in home to form a computational cluster for matrix multiplication. A PC with Core i7-4770K and 32GB of RAM, an ASUSPRO laptop with Core i7-7500U, a Raspberry Pi3 with ARM Cortex-A53, a Galaxy S9 smartphone with an Exynos 9810, and a S6 with Exynos 7420, and a Surface Pro with an i5 processor.

I am using Go for the programming language. It can run on Windows (for my PC and laptop), on linux (raspbian) and on Android, and I like working with Go. The premise is simple, I have two matrices A, B (for simplicity, square matrices) and I want to calculate their product. The challenge is how to distribute the task over our nodes, based on their respective speed and the speed of the network.

First, I do not incorporate all the nodes, I am only going to tackle the problem with 2 nodes. Multiplying 2 matrices is a process involves doing some products and sums, and each of these take some time, denoting the time for a single * operation with t_p and an addition with t_a the total time for NxN matrices would be:

$$t_{computation} = N^3(t_p + t_a) := mN^3$$

So the time grows linearly with respect to N^3, and the slope (let's call it m, it's the inverse of processor speed) is a property of the processor. We can expect higher slopes for slower nodes, also if we really take high N, we might clog the RAM and then swapping would incur additional burden and then this model will not be so accurate.

For 2 processor, I choose the PC and the laptop and try float32 matrices.

To test this model, we profile our multiplication loop, and plotting the results (for each value of N, time has been averaged over 20 trials. The numbers were pretty stable across trials: low variance.):

The results, and the high value of Pearson correlation coefficient show that our linear model for the computation time is accurate. We also model the network time linearly, we assume that in either direction the speed is the same. So for sending N floats around:

$$t_{network} = vN$$

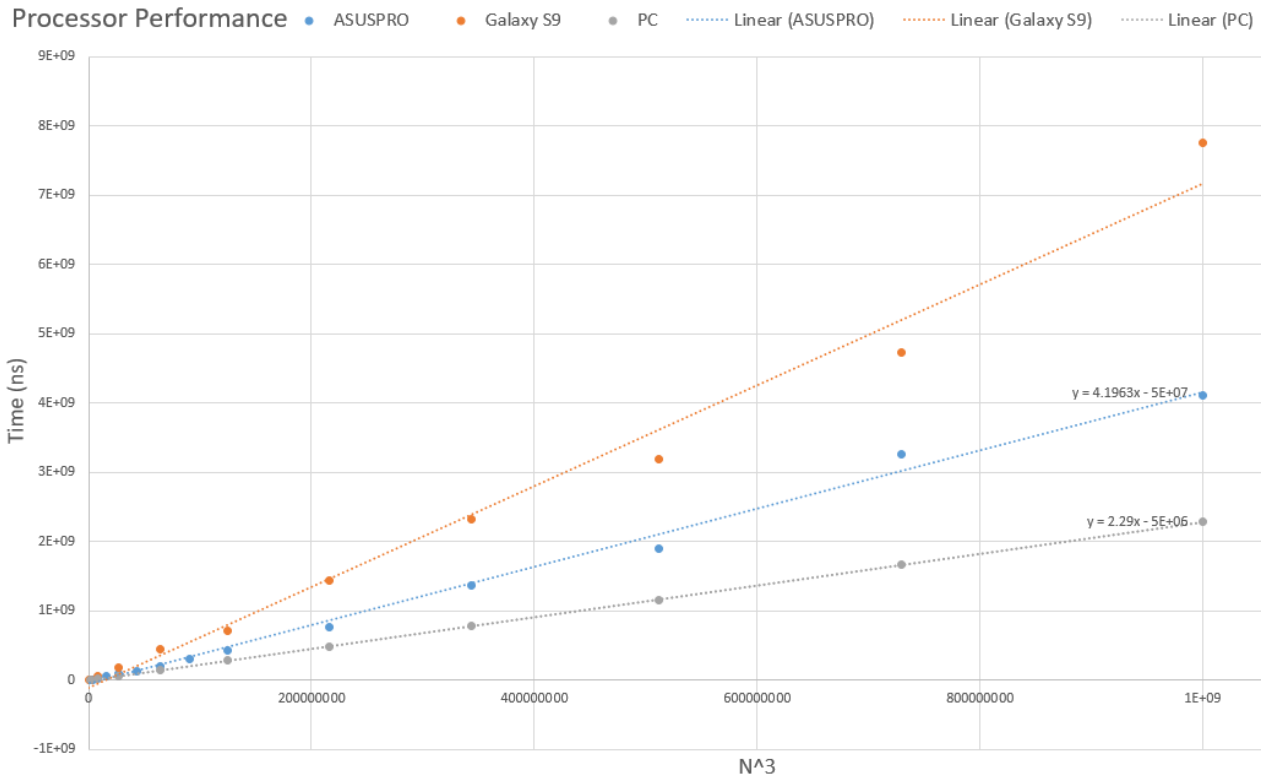

Figure 1: Processor multiplication performance

Again we test it out and plot the results. This time there was much variance across trials, and networking, even in a local network which I tried, was not reliable. But still the linear assumption proves fairly accurate.

Denoting the server (which initially has all the data) as node 1, and the remote client with 2 and With these models, if we split the resulting matrix C, into a rectangular region with eNxN and (1-e)NxN and task the remote client with the first chunk and the server with the second,we the chunk on the server would be available at:

$$t_{server} = m_1(1-e)N^3$$

And the chunk done by the remote client at:

$$t_{client} = m_2N^2(1+2e) + m_2eN^3$$

Obviously we would need to send less data over the network, if we choose the faster node as the server. The total time of the calculation is the maximum value between the times we calculated above, it is easily provable (as the two times are monotonic in e) that to minimize the time, we would need to choose e, such that the two times would be equal. (This is also intuitive, we would not want either node to wait for the other one to finish)

Solving the equation $t_{server} = t_{client}$ we get:

$$e = \frac{m_1N - v}{(m_1 + m_2)N + 2v}$$

It can be seen that as expected, if the network is very fast (v approaches 0) the optimal partitioning would be by giving each processor a chunk proportional to it's speed.

Based on the data we gathered from profiling the network and plugging into this expression for N = 1000 we get eN = 760 and experimenting with the system we see that indeed the data agrees with theory and we have the lowest time at eN = 760. Without doing the partitioning, the PC could compute the product at 2.2 seconds, the laptop at 4.1 but combined and with the eN above we manage to compute it at 1.7 seconds. Trying out eN values from 600 - 900 we see how at other values the time would be greater, and our partitioning is indeed less beneficial.

Of course for larger matrices, the partitioning would be even more beneficial, as predicted by the equation we derived earlier.