Concurrency in a distributed Gigapixel Image Viewer

Einar Kristoffersen
einar.kristoffersen@uit.no
University of Tromsø
Tromsø, Norway

Contents

1 Preface

This should be before contents... Thank following people: John Marcus Bjørndalen - advisor, bjørn - technical staff, ...

2 Summary

As the current implementation of the Gigapixel image viewer at the TromsøDisplaywall is inefficient, hard to maintain and unstable, the need for a new design is growing. This technical report will describe the new concurrent and flexible design of the Gigapixel image viewer.

lessons learned.

3 Introduction

3.1 Motivation

What is the motivation for this project? Maintaining the system more effective - concurrency

3.2 Description of problem

What is the problem I am solving?

3.3 Contents of this report

What will this report contain?

3.3.1 Requirements...?

What is the requirements?

4 Technical Background - Theoretical framework

Talk about the technicalities... Golang, concurrency vs parallelism,

4.1 The Go programming language

Go is an open source project developed by a team at Google as well as other contributors from the open source community. It was designed to address the problems faced in

software development at Google and became a great tool for engineering large software projects.

4.1.1 Why use Go?

As mentioned, one of the goals of this Project is to create a implementation that is maintainable. The current implementation of the image viewer is written by using a combination of several languages and as a result, it is hard to maintain the code. There was a discussion of which programming language would be best suited for the task. Both Python and Go was solid candidates in order to create a maintainable implementation, but the decision landed on the use of Go to perform this task. Unlike Python, (C and C++), the Go programming language is build for concurrency. Go has the advantage of goroutines..

4.1.2 Concurrency build on CSP

Both concurrency and multi-threaded programming have a reputation for being difficult to use. Partly because of complex designs such as pthreads and partly too much focus on low-level details such as mutexes, condition variables and memory barriers. Higher level interfaces enables much simpler code and have been more successful in the past. One of the most successful models for providing high-level support for concurrency comes from CSP, by Tony Hoare. Channels... This is why the concurrency is build on the idea of CSP...

4.1.3 Goroutines instead of threads

Goroutines is a structure of behaviour that makes consistency easy to use in Go. The idea is to multiplex independently executing functions, coroutines, onto a set of threads. When a coroutine blocks, the run-time moves the other coroutines on the same os thread onto a different, runnable thread so they can keep running and not be blocked. The programmer sees none of this, which is exactly the point and makes it very easy to use the interface provided by this language. The result, also called a goroutine, can be very cheap. There is little overhead other than the memory for the stack, which is just a few kilobytes. In order to make the stacks as small as possible, Go uses resizable bounded stacks. Goroutines are given a few kilobytes, which in most cases is enough, but when it isn't the runtime grows and shrinks the memory for storing stack automatically. This allows many goroutines to run in a small amount of memory. Because of this, you can easy create houndreds of thousands of goroutines in the same address space. If a goroutine was just a thread, the system resources would run out at a much smaller number.
https://golang.org/doc/faq#goroutines

4.2 Concurrency

4.2.1 Concurrency vs. Parallelism

When people hear the word concurrency they often have a tendency to think about the concept of parallelism, a related but quite different term. In programming, concurrency is the organization of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. When you run a program implemented by the use of concurrency, the executing processes is overlapping each other. This means that operations by different processes can work simultaneously, but they doesn't necessarily start at the same time. Parallelism often refers to the technique of making programs fun faster by performing several computations in parallel, literally at the same time. This requires multiple computing units, while the concurrency often refers to techniques that makes a program more usable and requires at least one processing unit. Concurrency can be parallelism, but not the other way around. Concurrency more powerful than parallelism.
https://blog.golang.org/concurrency-is-not-parallelism

5 Methods

6 Requirements specification

7 Design

The image viewer is designed with a master-slave solution, where the master is either the frontend node or a laptop and the slaves is the 28 tiles in the cluster. An image is

7.1 Display Tool - Keyboard input

7.2 Draw/Blit

calculate start position, bounds – only draw images with a position inside the tile's bounds

7.3 Data transfer

marshal, unmarshal, send, receive

7.3.1 Program flow

how it works - shortly

7.3.2 Data flow

about the rates
Whatever kind of input method is chosen, the received input will change the position variables kept at the master. Only when at least one of these variables are changed, they all will be marshalled/packed into a state. Every time a new state is paced it will be inserted into a buffer/queue (FIFO) where it will be sent out as a multicast to the slaves periodically.

Channel as buffer: Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

when a channel is full, the sender waits for another goroutine to make some room by receiving you can see an unbuffered channel as an always full one : there must be another groutine to take what the sender sends.

Variables: length of buffer/queue sendrate – how often should we let it send, FPS... empty the whole queue or just a bit?

We do this because we want the movement of the image to be as accurate as possible according to time with a minimal latency. We can afford to loose some states in this solution, compared to a solution where you send all states, obtain a huge latency over time and cannot afford loosing any states.

Input rate The input rate is the amount of received user input during one second. This is at its peak when the image is constantly moving.

Variable change rate The amount of times the position variables (x, y and z) is changed by the user input during one second will from now on be referred to as the "vcr" (variable change rate). Depending on the vcr is high or low, it might create a bottleneck in form of a queue before a state is multicasted to the slaves.

Send Rate-muticast The send rate is the amount of multicasts the master can complete per second. As the master will send the current state of the position variables periodically, it will have a certain send rate. If the send rate is lower than the vcr, the send queue will be filled up and a latency will occur. As the queue grows larger, the time from receiving user input at the master to the corresponding state is sent from the master also increases. Not dealing with this problem will result in a growing latency and (show graph) from tests... To deal with this problem we introduce a empty policy saying that if

the a state is tried to be inserted into a full buffer or the buffer grows too large, we empty the whole buffer.

blit rate Flow charts

7.3.3 Network flow

A big design choice is to choose which way to send data out to the slaves. Several ways was considered when making this choice.

Sequential solution The sequential solution is based on the master sending only one request to a chosen slave. Depending on how this solution is implemented, the slave forwards this request to some or all of its neighbours in vertical, horizontal or both directions. By introducing a rule about not blitting or forwarding a state that equals the last received state, a slave receiving two or more of the same request will not contribute to spamming the other slaves with the same request over and over again. To make this solution most effective the master should choose the tile that is closest to the center due to the configuration.

The forwarding of requests results in a queue of slaves where the first slave is the one receiving from master and the last one is the one furthest from the center. The slaves between the head and tail of the queue is a collection of the slaves forming the shortest route between these two. This approach is said to be sequential because the last slave in the queue must wait until all the salves before it has received and forwarded the request, one by one, to the next slave in the queue.

Unfortunately this approach comes with a few problems. Even if the different queues send their requests concurrently after the head has received it from the master, each slave must wait for the request until the slave above has forwarded it. With a display configuration of 7 tiles in width and 4 tiles in height (maximum amount of tiles), the amount of sends will be 4 after the head received the request from the master. This causes a big latency from master sends the request to the last slave receives is.

Another problem is because of the latency this approach causes, the spamming rule we introduced will only work to a certain degree. If a slave S receives requests from more than one neighbour, let's say A and B, then this can be the case: S receives the request r1A, followed by r2A and then r1B.

What happens is that the delay causes S to receive two different requests from A before the first request is received by B. Since the last request received before r1B was r2A, S will both blit and forward this request twice.

Because of the concurrency, this is not a "clean" sequential solution. That would be to let the master send one request to one slave and let each slave forward only one request further. This in not even considered as an alternative since the last slave in the queue

has to wait for all other slaves in the configuration to forward the request. Forwarding the request as much as 27 times would create a even greater delay before the last slave receives the request.

Multicast solution The multicast solution is quite simpler. The master sends the same request to all slaves by looping over the ip addresses and starting goroutines that send the request to each of them. This is technically not a multicast since the sending of requests is done concurrently, but it is a multicast like solution.

This will still result in a little delay from when each of the slaves receives the request, but this delay is so little that other factors will have greater impact on the total delay from a request is sent to a request is blitted on the screen. E.g. network traffic at each of the slaves or latency from fetching an image.

8 Implementation

9 Tests/Evaluation

10 Results

11 Discussion

11.1 Why SDL?

SDL, OpenGL, Azul3D...

11.2 Further works

in the future...

12 Conclusion

As we can see from the results and evaluation of the implemented design ... lessons learned...