Anh Tran & Nathaniel K. Blanquel
CPS 376
Prof McDanel
12/17/21

Distributed Image Processing with OpenMP and KissNet

Image processing is a popular area to explore in computer science, it is also an area that is computationally intensive, and applying different filters/algorithms to an image takes more time as the size of the image is increased. By building a distributed system (built using KissNet networking library) that utilizes OpenMP [1] (a threading library), we can distribute the work of image processing across several machines or "worker nodes" and aggregate the result of all the work in the "master node" which will output the final processed image. In total, our system utilizes a total of five nodes, one a master node, and the other four the worker nodes. Furthermore, the worker nodes themselves use OpenMP to further parallelize the work that is done by making use of any further cores the nodes may have.

As mentioned previously our distributed system has a master node and four other worker nodes (see Figure-1). The master node handles the slicing of the image and the work scheduling. Image slices produced by the master node will be sent to the worker nodes, and each worker will perform an image processing algorithm on its assigned slice and pass the result back to the master node when done.

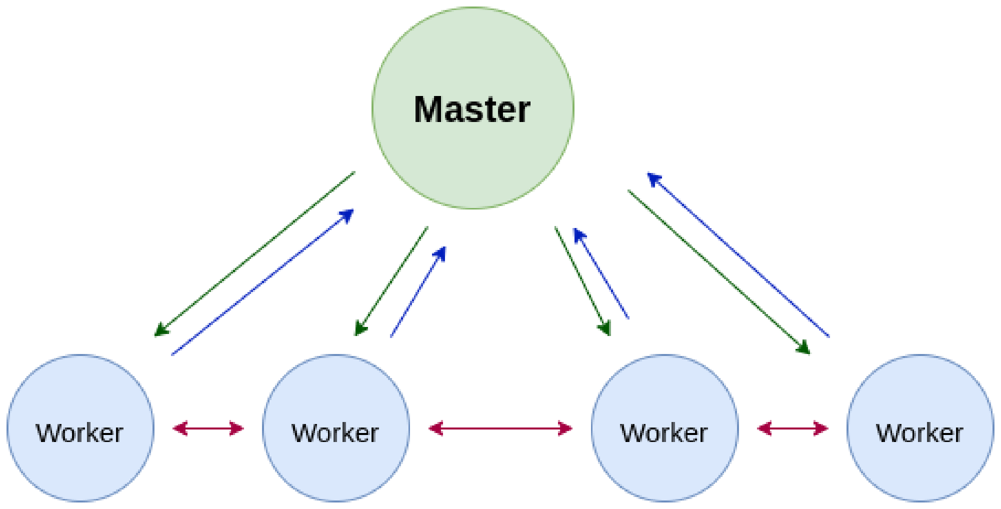

Figure-1, our distributed system, green represents transfer of metadata and slice, blue represents transfer of processed slice, magenta represents sharing of information between workers for thresholding (later removed and made part of metadata transfer)

The work scheduling for our distributed system is pretty basic. For each worker we have a flag to determine whether it's busy or not, if it's not busy then we pop a image slice off a vector containing all slices to be processed, send it to the worker, and set the flag to be true (to represent that it's busy). A worker's flag is set to false (to represent that it's not busy) when a "Done" message is received from the worker. The master uses OpenMP to asynchronously listen for "Done" messages from the workers while it checks for any available workers and sends further slices to them.

A big challenge of this project was handling all the back and forth communication going on between the master and worker nodes. To handle all our communication we used the KissNet library. Our distributed system makes use of KissNet by setting up tcp endpoints at specific ports (3000, 3001, 3002, 3003). Then we also use buffers as means of catching all data coming in to either the worker or the master. Each buffer can hold a limited amount of bytes which proved to be a challenge of its own in terms of setting up the communication. This first challenge was that large enough images wouldn't fit in only one buffer (which we set to hold 4096 bytes at a time). To take into account for this, each worker node runs an infinite while loop and within it we have different branches with different flags that determine what data we're fetching and what to do with the data we get on the worker side, and because we're running this in an infinite while loop we can continue to read in new buffer data until we are sure that the image transfer has been complete. At which point the worker will perform the requested image processing algorithm.

Apart from and before we send the image, we send metadata on the image to the worker via a csv string. This csv string is parsed differently depending on the op code present in the string. The string takes the form of "*w,h,op,id,...xxxxxxxxx*". Putting all our metadata into a string allows the master node to send it to the worker all in one go. The x's are there as a way to ensure that the buffer used to read the metadata (of size 55) is full. This is important to make sure of because the buffer that's meant for the metadata might catch part of the image if not full which would lead to the worker not being able to catch the whole image and causing it to stay in its current state in the infinite while loop instead of proceeding to do any image processing. The x's can be overwritten with extra metadata if need be. After the worker has received the metadata string it parses out the various information in the string and uses the data accordingly to set the width and height of the chunk to be processed and sets the operation code used by the worker to determine to either threshold (1), blur (2), or upscale (3) the image chunk. Once this metadata has been parsed we have the flag verifying the metadata is present so that our loop now changes to the image fetching stage as described previously. Once that's done, the worker proceeds to do some actual image processing.

Using the opcode either the threshold, blur, or upscale function is performed. Upscale is the simplest of the three, one only needs to multiply the width and height of the output chunk by the upscale amount passed in with the metadata and use loops with OpenMP to fill up the upscaled areas. For blur (box blur to be exact), the edge cases are tricky and although we didn't get it to work for our project the idea is simple. To handle the edge cases one needs to send additional padding (the height of the box / 2) with the image chunk, either on the top, bottom, or both. For threshold we also need the sum of all the pixel values of all the chunks, originally we planned to have the workers talk to each other and communicate their respective sums like in (Figure-1), but what we did instead is we calculated the sum as we sliced up the image and passed it along with the metadata. That way we avoid having to program further communication between all the workers, avoiding further complexity in our system.

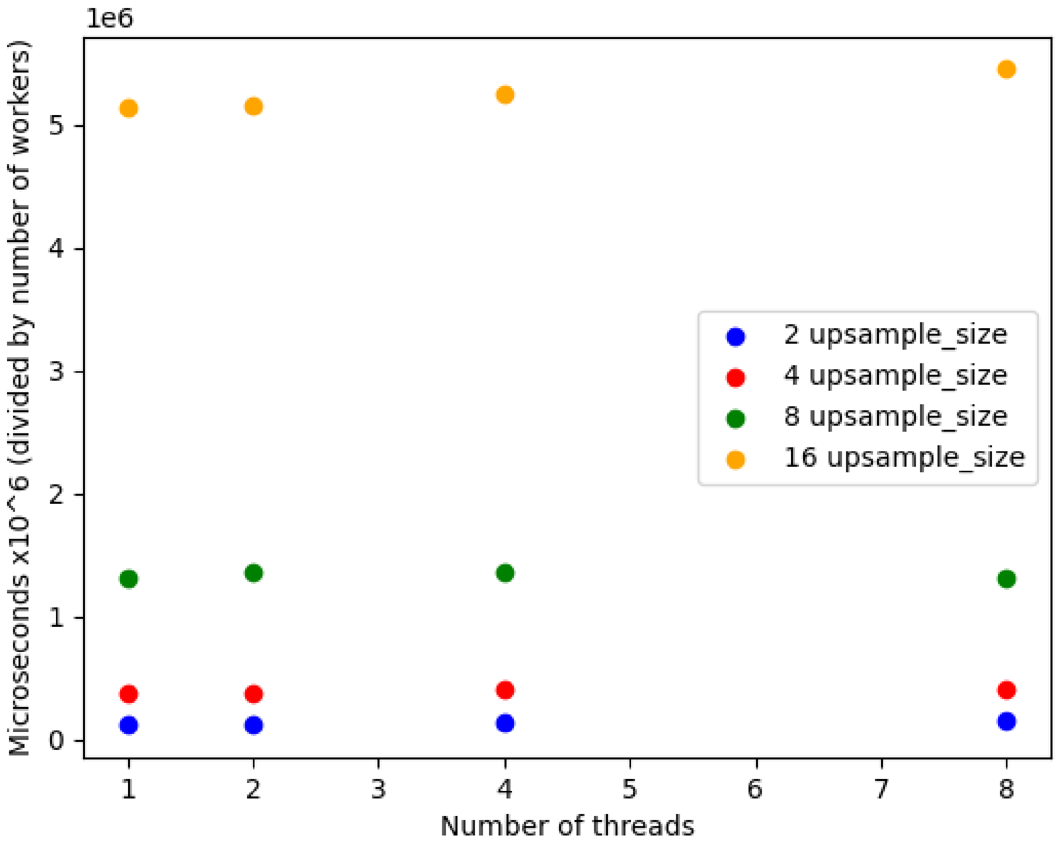

Figure-2: measured time (in microseconds) from sending out the image slices for upsampling in worker nodes to combining processed slices into result with different number of threads and upsample size

We did a benchmark of our system by running the upsampling algorithm on a 1600 pixels by 880 pixels source image using different values of upsample size (2, 4, 8, 16) and threads (1, 2, 4, 8), and a fixed number of 16 slices of the source image to send for the 4 workers. Then, for each configuration, we measured the time from when the master sends out the slices of the image to the workers until all the slices have been processed and combined into the resulting image. Since our system runs locally on one computer only, we divided the measured time by the number of workers, which is 4 in our case, to get the theoretical time one distributed computer finishes the job. The benchmarking result is shown in Figure-2. According to the figure, increasing the number of threads does not produce a faster execution time and that applies to every configuration of upsample size that we tested. We speculate that this is because our system consists of local workers communicating with the localhost (127.0.0.1) through different ports, and the overhead from networking affects the process of sending and receiving image slices. Furthermore, the image is already so big, upsampling it makes it even bigger, which takes a great amount of time sending through the sockets on the localhost. Another reason might be that since we use OpenMP to parallelize the sending of image slices in the master while having a thread for receiving images on the master end for each worker and we also run each worker separately on one machine to do its own sending, receiving and using OpenMP to process the image slices, we might not have been able to parallelize everything and thus some threads might have to wait for a long time for their turn to do their work, which explains why adding more threads does not increase the performance. We think that if we were able to run our system on an actual distributed system with multiple computers, we would be able to clearly see the difference in performance that we initially aimed for. Future improvements that could be made to this system would be to properly send the padding required for blurring as well as running the workers of this system on separate machines.

References

[1] *OpenMP*, 23-Sep-2021. [Online]. Available: https://www.openmp.org/. [Accessed: 17-Dec-2021].