TDDC78 - Report Lab1/Lab2 - loibi806 - goubh275

Averaging filter

To perform the averaging filter in parallel, we decided to split the image horizontally. Each node (in the case of MPI in Lab1) or each thread (in the case of pthreads in Lab2) will work on a different horizontal section. We first start by doing an horizontal blurring for each pixel in the current section, and then we do a vertical blurring.

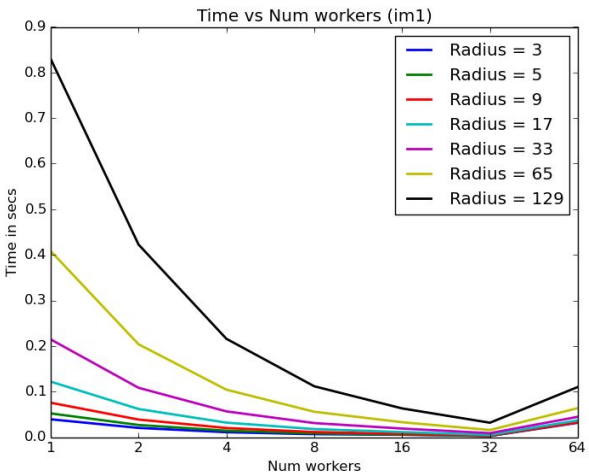
Implementation with MPI

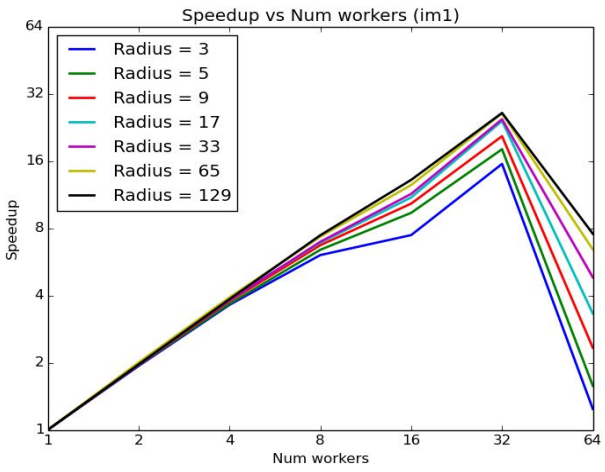
The first thing to do is to scatter the complete image to the different nodes, and this can be done using *MPI_Scatterv.* For this, we had to create a custom MPI type that was able to store a pixel, and we also had to compute for each node what rows it should work on. This information was stored into the *counts* and *offsets* arrays, then used inside the call to *MPI_Scatterv.* All other data, such as the size of the image, was sent to all the nodes using *MPI_Bcast.* Each node could now perform the horizontal blur.

After this, each node had to send its top and bottom rows (for a total of *2 x radius*) to its neighbors, so they would have the correct updated pixel values to perform the end of the algorithm. SInce the radius may be bigger than the size of the image's section given to a node, we sometimes have to send data from multiple neighbors.

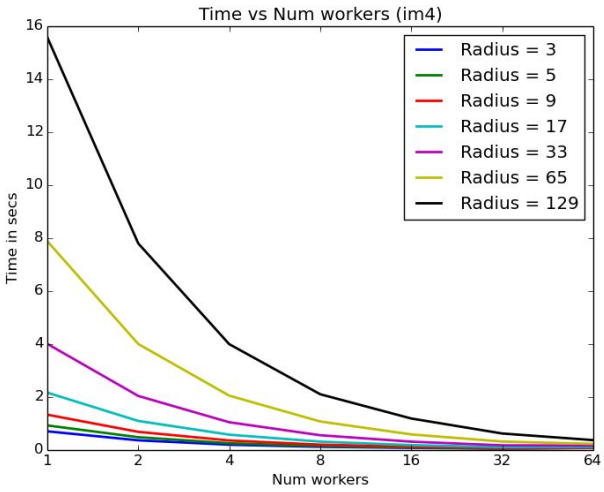
We then could perform the vertical blurring, with this newly obtains data for the edges of the section. Once it is done, we just need to gather the data from each node to the main node, that will then take care of writing it back to a file.

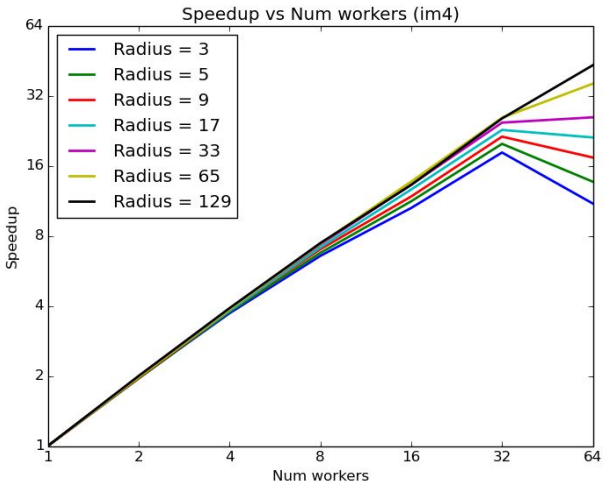
Results





For the smallest image, We see that doubling the number of workers approximately divides the running time by two, up the a certain number of workers, in this case 32. It is because at some point, the cost of creating all the workers and sending data between them is higher than the benefit created by more parallelism. For small radiuses, we this that the speedup becomes smaller after only 8 workers, because a small radius implies less computations, and therefore the ratio between the computation cost and the communication cost is smaller, reducing the effect of parallelism.



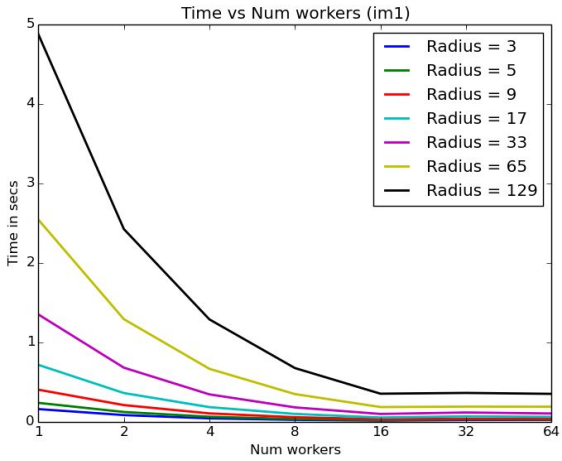


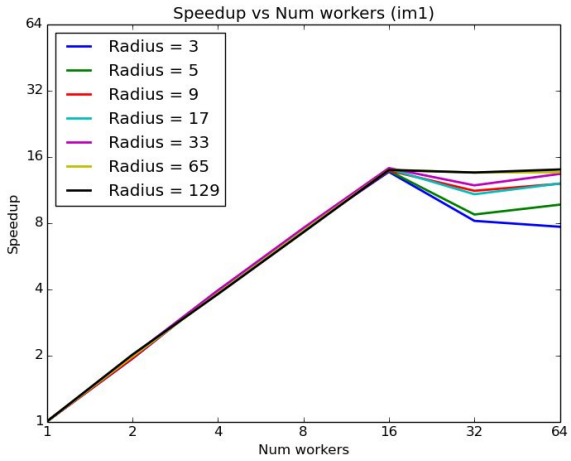
With a bigger image, we see that the speedup persist longer when increasing the number of workers, and it even continues to grow after 32 workers for radiuses greater than 33. Again, this is because we have a lot more of computations to do, compared to the time needed for communication.

Implementation with pthreads

SInce pthreads uses a shared memory model, we didn't have to scatter the data to each threads. We only needed to give each thread the information of the range of rows it should work on, and this is done when create the thread by giving it a data structure with all needed information inside. We also had to make sure that no thread started the vertical blur before they all had finished the horizontal blur, and for that we use a *Barrier,* that guarantee us that every thread has reached this point before allowing them to continue any further. Since each thread writes to different memory addresses, we didn't have any synchronization issue. In the end, no gathering is needed since we used a shared memory model.

Results





Using pthreads, we can still see that the runtime is divided by approximately two everytime we double the number, up to 16 workers. This is because pthreads can only use threads on a single node, and in Triolith there is 16 threads per node. For bigger images, the result is similar.

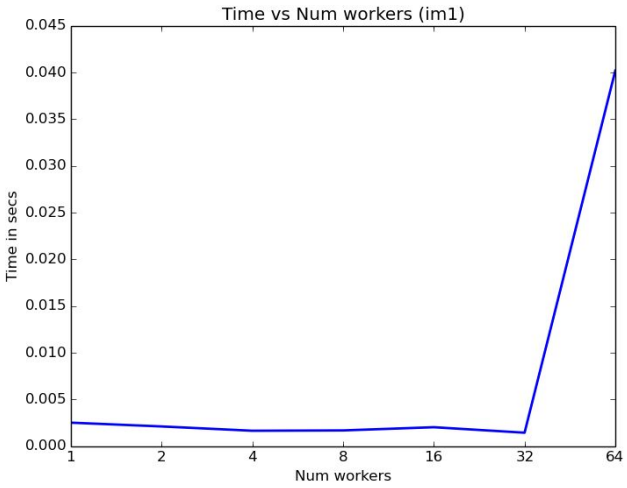
Threshold filter

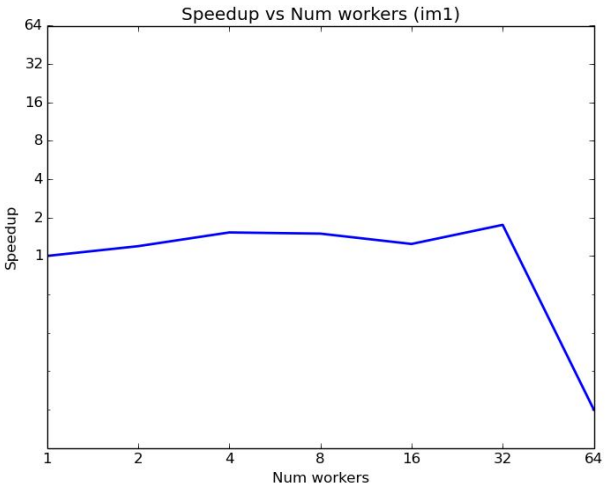
To perform this filter, we once again split the image horizontally. Each node/thread would start by computing the sum of the value of all pixels in its region. This sum would be added to the sum of all other regions, and used to find the average value of the pixels. Each node/thread could then use this average value to perform the threshold on its own region, by setting pixels to black or white depending on its value.

Implementation with MPI

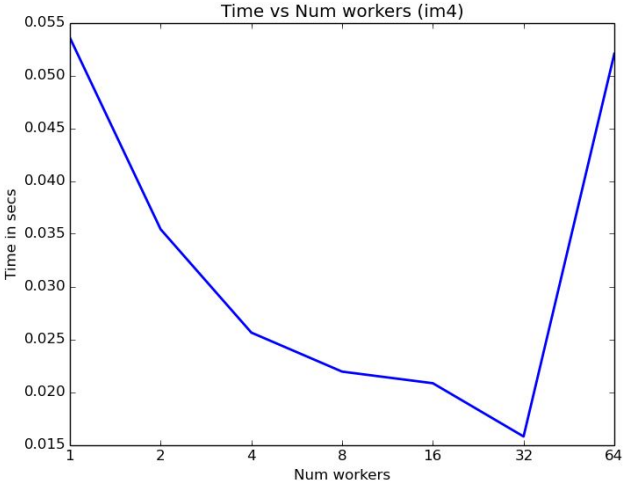
As for the blur filter, image data is scattered on each node using *MPI_Scatterv*, and other variable using *MPI_Bcast.* Each node will then compute its local sum, and all of these local sums will be put together using *MPI_Reduce* with the *MPI_SUM* operation. This will result in the node number 0 having the total sum of all pixels. This node will compute the average, and send the result to all the other nodes using *MPI_Bcast.* Now, each node will have the average value of pixels and can perform the threshold itself on their own data. At the end, as before, the image is put back together using *MPI_Gatherv.*

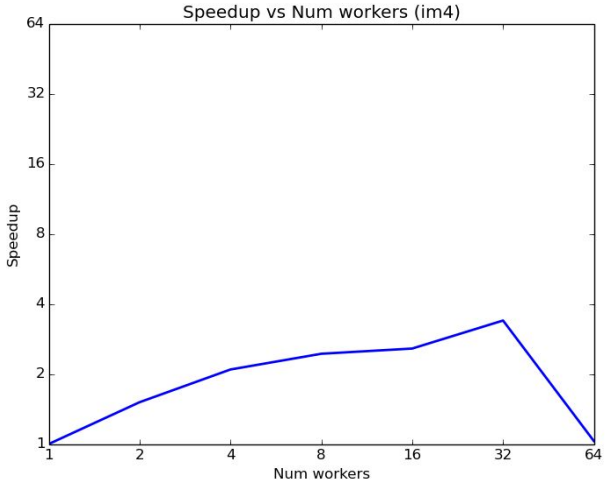
Results





With the small image, there is almost no speedup. This is because the threshold filter requires a very small amount of computations, and therefore the time needed to create and communicate between workers completely cancels the speedup due to parallelism. And after 32 workers, the cost associated to communication completly





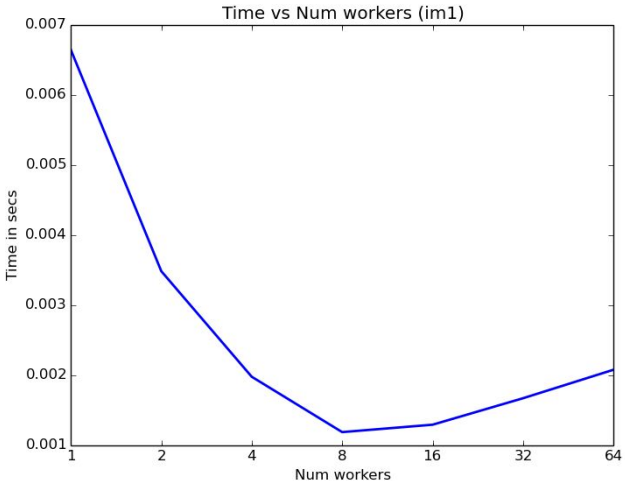
With a bigger image, this effect mostly disappear. It is still not a one to one speedup, but it is better.

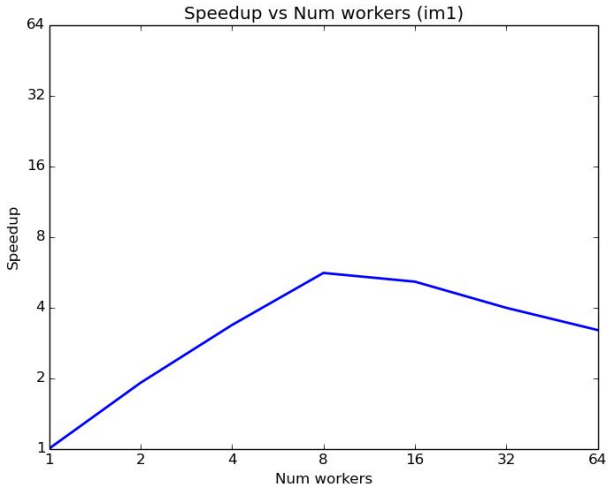
Implementation with pthreads

Again, we don’t send the image data to each thread but only information on the bounds each threads must work in between. We also send a pointer to an int called *sum,* that will be used as a common variable used by each thread to accumulate its local sum. Once each thread has computer its local sum, it will add it to this global sum, using

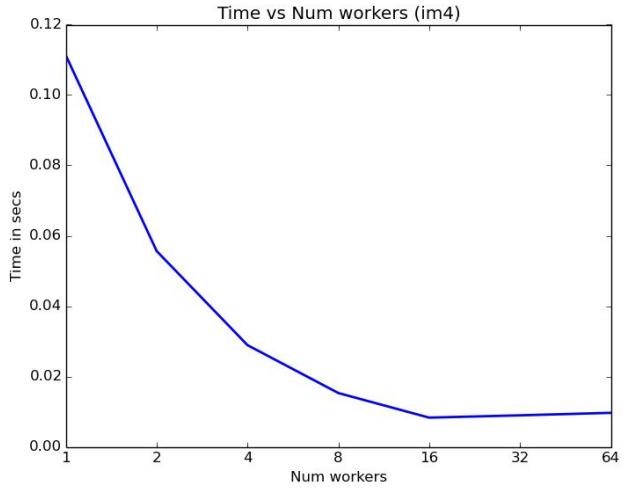
pthread_mutex_lock and *pthread_mutex_unlock* to ensure synchronized access to this global sum. Once each thread has added its local sum to this value, they can do the thresholding (This is ensured using a Barrie with *pthread_barrier_wait*). Again, no gathering is needed since pthreads uses a shared memory model.

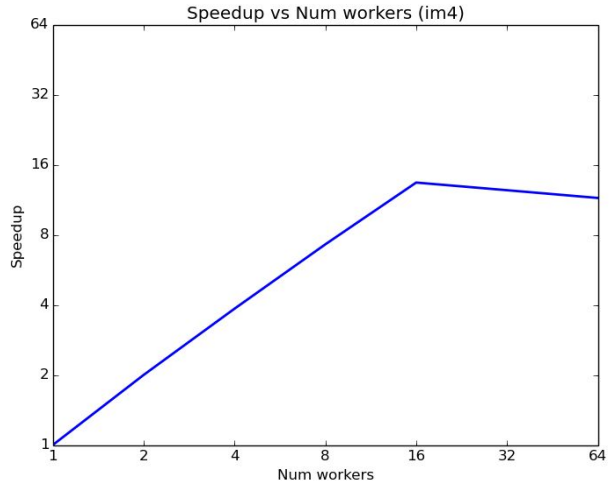
Results





For the small image, we see that the limit number of workers for speedup is 8, even if pthreads can use up to 16 threads, simply because we don't have enough computation to perform to make it worth the added thread creation cost.





With a bigger image, we manage to get up to 16 threads, and therefore using all the potential offered by pthreads.

Conclusion

If we want to notice a improvement in computation time using parallel techniques, we must have a big enough number of computation to compensate the cost associated with creation of threads/workers and communication. We also see that for two problems we had to solve, pthreads was more effective than MPI. We can explain this because the problem we had to solve didn’t require a lot of synchronisation, since each thread was working on different memory addresses, allowing pthreads to have almost no time waiting. With MPI, all the image had to be send back and forth between different workers, and this takes a lot of time.