Parallel Computing Of Image Transformation and Pearson Correlation Softwares

DV1567 - Performance Optimization,

Group 4

1st Jonathan Olsson
Blekinge institute of technology
Karlskrona, Sweden
Joos21@student.bth.se
19950201

2nd Lokesh Kola
Blekinge institute of technology
Karlskrona, Sweden
lokl20@student.bth.se
20000717

I. Blur Filter

A. Data partitioning concept

To enable parallel computing of the blur filter algorithm it was required to find areas of the given source code that could run independent on each other. This was done by investigating the overall architecture of the software and the desired algorithm to find out if the implementation should be done on a macro or micro layer. Once these areas was investigated, it is then possible to divide up the workload upon multiple threads that could perform the desired computations in parallel. By doing so, the available resources in a given system could be utilized better, which might allow the system to make the same amount of computations in a less period of time. This might have a direct impact on the total execution time of a given software. The blur filter algorithm is applying a blurring filter on an image of size $area = x * y$, where each area point consists off three different colors, red green and blue. Since the desire is to split up the workload against multiple threads, the idea was to divide up the total area upon the selected amount of threads that is supposed to do the work. In Fig.1, there is an example of the division of labour when the algorithm is run with four threads. As could be seen, each thread is given a start and stop interval where one thread is supposed to operate within. The idea is to start traverse from the start location and move through all the values in a column and then move onto the next column, until the stop location is reached. This is then repeated for each thread to make sure that all of the area points is covered in the image. The area of which a thread is operating in is adjusted based on how many threads that is created. By making each thread to only work on a portion of the image, there is no requirement to add additional overhead synchronization such as mutex locks to prevent the threads to write to the same place in memory, which might cause data corruption. In addition, extra overhead functionality might slow down the system since more instructions has to be performed.

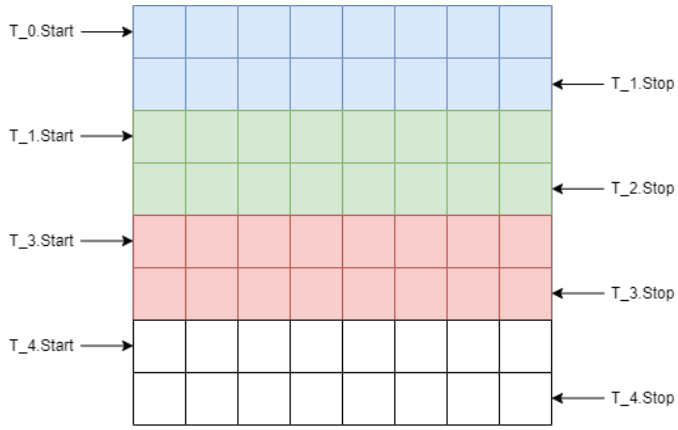

Fig. 1. Shows the division of labour for the given threads for the blur algorithm.

B. Code changes

First, the blur software inherit the performance optimization techniques that was implemented during previous assignment. Then it was found out that an additional improvement could be done by moving the Gaussian get weight function outside of the nested for-loop to make it independent on the size of the image. This change is presented in Fig.2. However,

```
Gauss::get_weights(radius, w);

for (auto y { 0 }; y < sizeY; y++) {
   for (auto x { startX }; x < stopX; x++) {
```

Fig. 2. Shows the code change for the first iteration.

some modifications was performed to the architecture of the file. First two new methods was implemented. One for running each application in sequential, without creating a thread. This was done to limit the extra overhead costs that is required to create and initialize one or multiple threads. This function inherit the same structure as the one that was presented in previous assignment. The second function has a generic function that allows the algorithm to operate in a given

interval. This interval gets adjusted by applying a dynamic range to the x-axis. To prevent race-conditions, the algorithm is only reading from the input image. In addition, the each thread is assigned their own matrix which they could operate on, in their private stack to speed up the access and to prevent data corruptions. Since the threads is operating in intervals, each of the thread could safely write to the final destination without having to worrying about other threads writing at the same time. This enables the software to work without mutex locks. The application logic, first initializes variables that is required independent on if the software is running with one or multiple threads. Then to reduce unnecessarily work, the application checks whether one or multiple threads has been selected. If one thread is selected, then the necessarily data gets sent to the function which will run it sequentially without creating a thread. If the case is that multiple threads has been selected, then the first step is to create a struct that could pass the required data with the thread to the function. Then the offset of the x-axis gets truncated to have a rough estimate on how large the intervals is going to be. Then the data that is going to be included for each thread is computed. First the intervals is computed, the total size of the x and y-axis gets included with the desired radius. Then the reference point to the input matrix and the output matrix is stored to reduce the amount of copying. For some corner cases where the size of the image is uneven, it could be the case that the intervals get unfair distributed, since by truncating the intervals will make the last thread have additional work compared to the other threads. To shed light on the problem, please take a look at the following example where the size of the x-axis is 23 and the intervals is divided upon five threads (1):

$$23 = 4 + 4 + 4 + 4 + 7 \tag{1}$$

As could be seen, the last thread has to do almost 60% more load compared to the other threads. To mitigate the effects of this phenomenon, the idea is to compute the difference between the xSize and the last interval position. Then, since the distance will always be ranging from values larger than zero to slightly under the chosen amount of threads, it is possible to iterate through all threads and increase the start and stop intervals by one. Once this is done, it is required to make sure that the intervals still operate within the boundaries of the array, this is done by setting the start position for the first thread to zero and the last thread to inherit the size of the x-axis. By applying this technique, the load distribution will instead look like the following example (2):

$$23 = 5 + 5 + 5 + 4 + 4 \tag{2}$$

The source code for the application logic could be seen in Fig.3

C. Performance

Fig.4 presents the changes of execution time from earlier iterations before the parallel computing took place. As could be seen, the time reduced from approximate 16 seconds, down to 2,6 seconds. In Fig.5 it is possible to see the execution

```
if(threads==1){
 calcBlur(xSize,ySize,radius,&dst,&writeMatrix);
}
else{
//Matrix scratch { PPM::max_dimension };
struct BlurDataPasser intervals[threads];

//initialize structs and calculate offsets
int offsetX = xSize/threads;
for(auto i{0};i<threads;i++){
intervals[i].startX = offsetX*i;
intervals[i].stopX  = offsetX*i + offsetX;
intervals[i].sizeX = xSize;
intervals[i].sizeY = ySize;
intervals[i].radius = radius;
intervals[i].m = &dst;
intervals[i].writeMatrix = &writeMatrix;
}
//load balansing
unsigned diff =xSize-intervals[threads-1].stopX;
if(diff>0){
    for(auto i{0};i<diff;i++){
        intervals[i].startX++;
        intervals[i].stopX++;
    }
    intervals[0].startX = 0;
    intervals[threads-1].stopX = xSize;
}
```

Fig. 3. Shows the source code for the blur application logic.

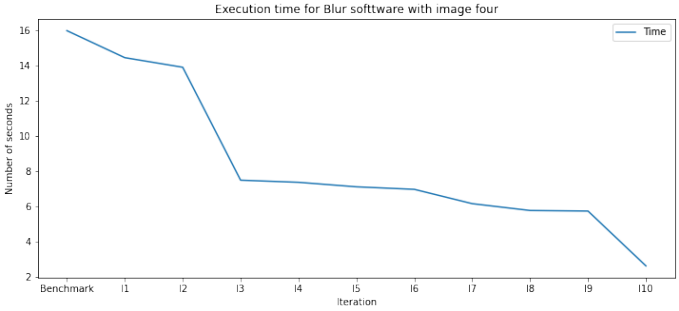

Fig. 4. Shows the execution time from earlier iteration.

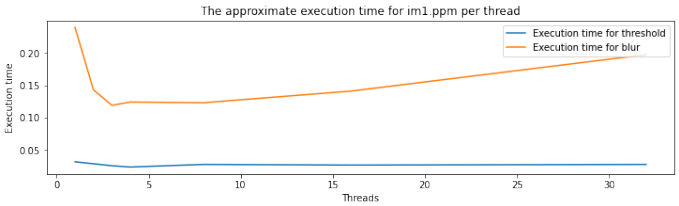

Fig. 5. Shows the execution times for im1.

time when executing the blur and the threshold algorithm with various amount of threads. As could be seen, there seems to be clear advantages speaking of execution time to roughly four threads, then the performance decreases. This is probably due to that the SUT only has four cores and creating more threads than this seems create costly overheads. For the threshold algorithm, the line is almost flatline. This indicate towards that since the algorithm was so simple and the size of the image was rather small. The costs for making additional threads seems to neglect the benefits of running the algorithm in parallel. This remain true for each of the Figures 6,10 and 11. However, it seems that the extra costs for creating more threads than what the SUT is able to handle, will not have significant

negatively impact on the performance. This is probably due to that the amount of computations that is required is more time consuming than creating the overhead for the threads.

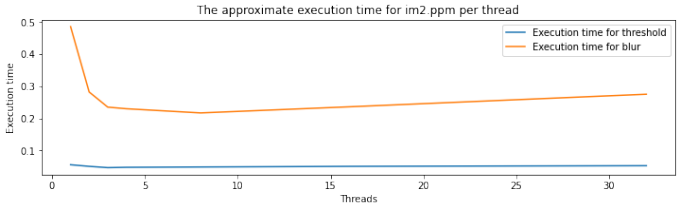

Fig. 6. Shows the execution times for im2.

II. Threshold Filter

A. Code changes

The procedure for finding where the parallel computing of the threshold filter took place, was the same as discussed in the blur filter section. For the threshold filter, the parallel computing was decided to be implemented on an algorithmic level and then modify the application logic to enable the software to run with one thread sequentially without extra overhead. In addition, the software should also be compatible to being run with two or more threads. The application logic that was used for the threshold filter is similar to what was presented for the blur filter. However, the data that is required to be sent to the function is different. The application logic also inherit the load balancing technique that was presented in previous section. A brief sample of the Threshold filter is presented in Fig.7.

```
if(threads==1){
 sum /= nump;
 calcThreshold(0,nump,sum,&dst);
}
else{
//Compute ranges.
struct DataPasser intervals[threads];
int offset = nump/threads;
for(auto i{0};i<threads;i++){
intervals[i].start = offset*i;
intervals[i].stop  = offset*i + offset;
intervals[i].sum = sum/nump;
intervals[i].m = &dst;
}
//Load balancing threads
unsigned diff =nump-intervals[threads-1].stop;
if(diff>0){
    for(auto i{0};i<diff;i++){
        intervals[i].start++;
        intervals[i].stop++;
    }
    intervals[0].start = 0;
    intervals[threads-1].stop = nump;
}
```

Fig. 7. Shows the source code for the threshold application logic.

B. Data partitioning concept

Moving over to the technique that was used to compute the workload in parallel, it applied a similar approach that was presented in Fig.1, but for one dimensional array instead.

The idea is to first compute the start and stop ranges for the x-axis. Then, since the algorithm is rather simple, the technique is just to iterate through the array within the given ranges and then determine if a given pixel should be set to either black or white. Due to the simplicity of the algorithm, the threads could read, operate and write to the same location without interfering with each other, since they work within excluded ranges.

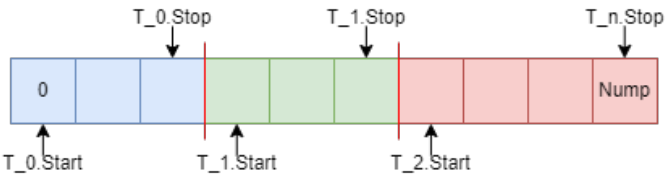

Fig. 8. Shows the division of labour for the given threads for the threshold algorithm.

C. Performance

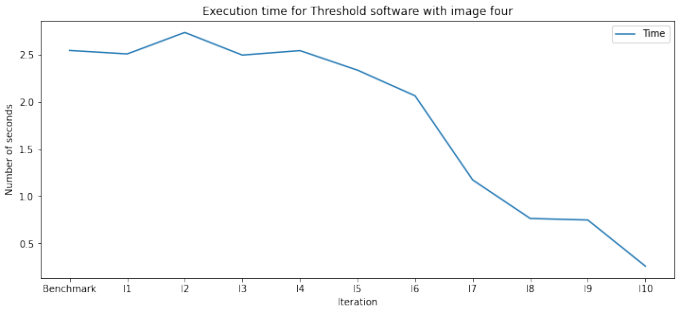

Fig. 9. Shows the execution times from earlier iterations.

Fig.9, presents the execution time for the threshold algorithm for each iteration. As could be seen, at the earlier iterations, there is some fluctuations that was captured by the measurement tool because there was no changes to threshold at these iterations, then at later iterations there was an improvement in execution time of approximate 10 times compared to the original version.

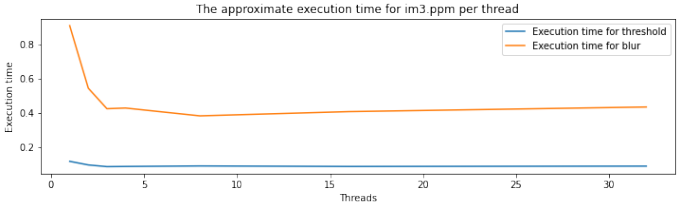

Fig. 10. Shows the execution times for im3.

III. Pearson correlation coefficient

A. Data partitioning concept

The heart of the pearson correlation coefficient software is built upon the algorithm (3). The complexity of the algorithm

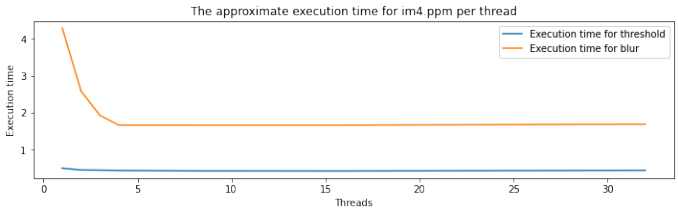

Fig. 11. Shows the execution times for im4.

will have an impact on the performance and should be dealt with care to avoid expensive work.

$$r = \frac{\sum_{i=1}^{n}(x_i - \tilde{x})(y_i - \tilde{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \tilde{x})^2 \sum_{i=1}^{n}(y_i - \tilde{xi})^2}} \tag{3}$$

By investigating the current implementation of the correlation coefficient method it was found out that the majority of the load is performed in the first rounds and is then decreasing for each round. The workload pattern could be seen in Fig.12. Due to the characteristics of the implementation, it is not possible to divide up the workload ranges as it was done in previous sections. Instead, a load balancing algorithm was developed to

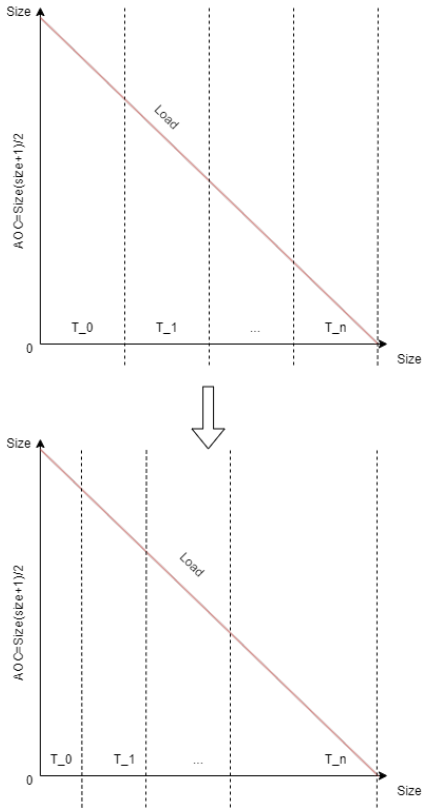

Fig. 12. Shows the workload concept for pearson calculations.

better balance the load toward multiple threads. The idea is to first calculate equal ranges by dividing the size of the dataset by the amount of threads(4) and then calculate an offset(5) that will shift the intervals to the left, which will impact the AOC. In addition, for large data sets greater than a size of 512 and for more than two threads, the offset ranges gets further shifted to the left to even out the AOC under the curves. In Fig.12 the above graphs shows the challenges with equal ranges and the below graph explains how the load was balanced out.

$$Range = \frac{Size\ of\ dataset}{threads} \tag{4}$$

$$Offset = \frac{Range}{threads} \tag{5}$$

The final ranges then got computed with(6),(7),(8) and (9).

$$start_{thread_i} = \sum_{i=0}^{threads-1} i * range - offset \tag{6}$$

$$stop_{thread_i} = \sum_{i=0}^{threads-1} i * range + range - offset \tag{7}$$

$$start_0 = 0 \tag{8}$$

$$stop_{thread-1} = Data\ set\ size \tag{9}$$

The offset will be used to move the load from the first thread and instead make the ranges for the other threads larger to even out the workload.

B. Code changes

The vector size got calculated with (10) to reduce unnecessarily copy instruction.

$$vector_{size} = \sum_{i=1}^{data\ set} x_i = Data\ set_{size} \frac{(data\ set_{size} + 1)}{2} \tag{10}$$

The estimated size for an arbitrarily vector for a given thread was calculated in similar fashion(11).

$$Thread_{vectorSize} = \frac{vector_{size}}{threads} \tag{11}$$

In addition, a comparison between the vector pushback, insert and emplace_back was done. According to the tests, the insert function performed the best for this given application. The application got compiled with the "-O3" flag to optimize the compiler for performance and the debugger flags got removed. The input and output buffers got their size increased to 8192 and there was also an expensive "endl" operation placed in the output loop, which got replaced with "$\backslash n$". If the reader is interested to see the implementation for the pearson load balancing algorithm and and pearson function this will be presented in Fig.13 and Fig.14.

C. Performance

In Fig.15 it is possible to see the breakdown of execution time when the benchmark tests was performed for the pearson software. As could be seen, the execution time steadily increases as the size of the input grows larger. For the largest dataset 1024, it took approximate 35 seconds to complete the task. It is expected that the execution time will grow with respect to the size, which will impact the scalability negatively.

```
//Calculate ranges
struct corrData intervals[threads];
int resaverad = factor1/threads;
int range = datasets.size()/threads;
int offset = range/threads;
 //Initalize threads.
for(auto i{0};i<threads;i++){
    intervals[i].start = i*range-offset;
    intervals[i].stop = i*range+range-offset;
    intervals[i].datasets=datasets;
    intervals[i].working.reserve(resaverad);
}


   //Squeeze the AOC to the left, optimizing for larger files.
 if(threads>2 && datasets.size()>=512){
  for(auto i{1};i<threads;i++){
      intervals[i].stop-=offset;
      intervals[i+1].start-=offset;
}

  }

  //fix the outbound ranges.
  intervals[threads-1].stop = datasets.size();
  intervals[0].start = 0;
```

Fig. 13. Shows Loadbalancing algorithm for pearson.

```
void* threadCorrelation(void *arg){
struct corrData *corrDP= (struct corrData*)arg;

    //Calculate pearson correlation coefficients
    for (auto sample1 { corrDP->start }; sample1 < corrDP->stop; sample1++) {
        for (auto sample2 { sample1 + 1}; sample2 < corrDP->datasets.size(); sample2++) {
            auto corr {pearson(corrDP->datasets[sample1], corrDP->datasets[sample2]) };
            corrDP->working.insert(corrDP->working.end(),corr);

        }
    }

    pthread_exit(NULL);

}
```

Fig. 14. Shows correlation pearson implmentation.

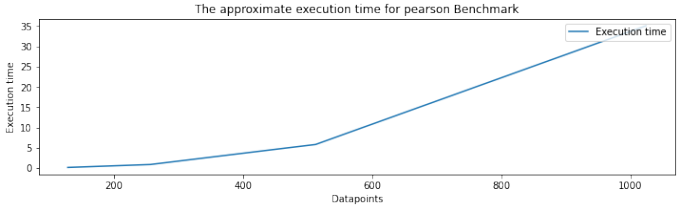

Fig. 15. Shows the benchmark tests for pearson.

Fig.16 breaks down the execution time for the final performance iteration of pearson. As could be seen for the data sets 128 and 256 there was only no benefits of running the application in parallel. The reason behind this might be that the amount of data that is being required to be computed is small and there is an overhead cost of creating threads. For the data set 512, there was a minor performance increase for up to four threads and then it stagnated. For the largest dataset it took approximately eight seconds to execute for running the application sequential. By increasing the amount of threads, a notable improvement was measured. For four threads and above, the execution time took approximately three seconds to complete. The reason behind the stagnation is that the System under test only supports four CPU. It is expected that an increase of CPU's will shorten the execution time. By comparing the benchmark tests with the optimized version, there was an increase of performance with 32 seconds or almost 12 times faster, when the application was run with four threads. It is expected that the application will scale better for larger data sets when the running the application in parallel, compared to running it sequential.

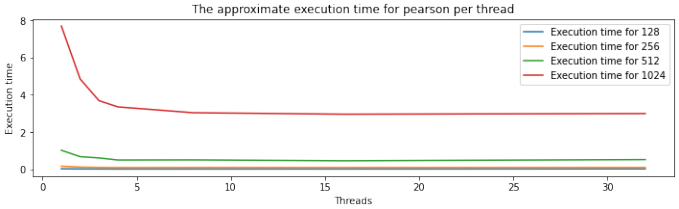

Fig. 16. Shows the benchmark tests for pearson.

In Fig.17 it is possible to see the CPU load balancing when the pearson application was run with eight threads with the 1024 data set. As could be seen, the load is distributed towards

01:43:59	CPU	%user
01:44:00	0	89,36
01:44:00	1	65,52
01:44:00	2	65,52
01:44:00	3	57,14
01:44:00	CPU	%user
01:44:01	0	98,28
01:44:01	1	100,00
01:44:01	2	77,19
01:44:01	3	98,28

Fig. 17. CPU load for pearson applicaion.

each CPU core in a reasonable good fashion for the first second as the threads start at 01:44:00 and then ramps as could be seen in the next time frame.

IV. Discussion

By introducing parallel computing into the algorithms seems to be advantageous. However, by changing the architecture of the software, which might also change the characteristics. This might impact the cache locality and could create new hot-spots or areas that could be improved. If there was a longer time frame for this given assignment, it would have been recommended to make more iterations of the software, similar to what was done in the previous assignment to boost the performance of the pearson application. In addition, by creating a better load balance algorithm, would probably further improve the execution time since each CPU core could work more efficient for the pearson application. Also, if the structure of the correlation function was changed so it be possible to insert at a desired location in a vector or array where each thread could operate freely in excluded ranges, then it would be possible to remove the merging $O(n^2)$ loop. Lastly, it might be worth investigating if the write to file function could instead save its output to a buffer and then only print the contents of the buffer once, rather than printing multiple times to the file. By doing so, might reduce the write functions.