**High Performance Computing Project Report**

**Parallel Implementation of Histogram Equalisation**

**Prepared By : Zarana Parekh (201301177), Charmi Mehta (201301432)**

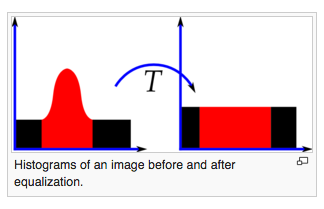
**Section 1 : Problem Description and Analysis**

**Problem Description :**

Histogram equalization is a method for modifying the dynamic range and contrast of an image by altering that image such that its intensity histogram has a desired shape.

The histogram of an image refers to histogram of the pixel intensity values. A mapping is done between the pixel intensity values in the input and output images such that the output image file contains a uniform distribution of intensities (a flat histogram). This technique is used in image comparison processes for detail enhancement and in correction of nonlinear effects.

The probability mass function and the cumulative distribution function of the pixel values in input image are used to determine the pixel values in output image. A transfer function is used to derive the required relation. The histogram and output pixel value computation can be carried out as 2 parallel steps.



**Histogram Equalisation Concept :**

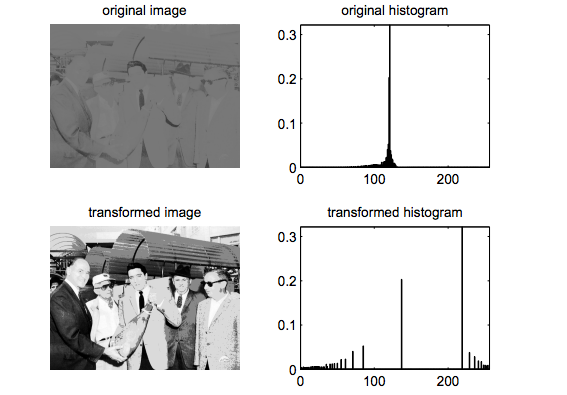
First we will calculate the probability mass function of the histogram of the original image

The intensity level n ranges from 0 to 255.

* + Then we calculate the cumulative Distributive Function of the Probability Mass Function.
  + This cumulative distributive function values is multiplied by the number of levels to find the new pixel intensities, which are mapped into the old values and our Histogram is equalized.

**Input and Output:**

* + Serial Program :
    - Input : A bmp image which is to be modified by applying the equalisation algorithm.
    - Output : Modified bmp image file.
  + Parallel Program :
    - Input : A bmp image which is to be modified by applying the equalisation algorithm and the number of threads
    - Output : Modified bmp image file.
  + The image sizes being considered are :
    - 128 x 128
    - 256 x 256
    - 512 x 512
    - 1024 x 1024
    - 2048 x 2048



Histogram Equalisation applied to a low contrast image

**Applications :**

Histogram equalization is often used to achieve better quality images in black and white color scales in medical applications such as digital X-rays, MRIs, and CT scan.

Histogram equalisation enhances the quality of the image and hence has applications in image processing. This technique is also used for enhancement of satellite images.

**References for further reading :**

* + *Parallel Algorithms of Histogram Equalisation on parallel computer of SIMD type*, TV Giang, published in Journal of Computers and Artificial Intelligence, Volume 10 Issue 5, 1991
  + *A novel parallel architecture for local histogram equalisation*, Mesrob I. Ohannessian, published in Visual Communications and Image Processing 2005

**System Configuration :**

|  |  |
| --- | --- |
| **Model name** | Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz |
| **CPU MHz** | 1200.000 |
| **Cache size** | 20480 KB for each processor |
| **cpu cores** | 8 |
| **fpu** | yes |
| **fpu\_exception** | yes |
| **cpuid level** | 13 |
| **cache\_alignment** | 64 |
| **address sizes** | 46 bits physical, 48 bits virtual |

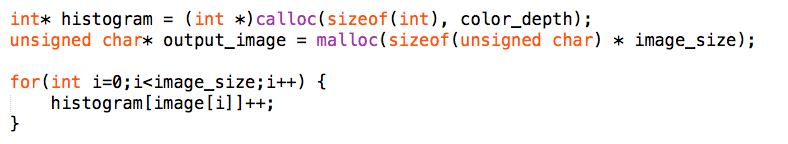
**Section 2 : Discussion of the Serial Algorithm**

**Pseudocode :**

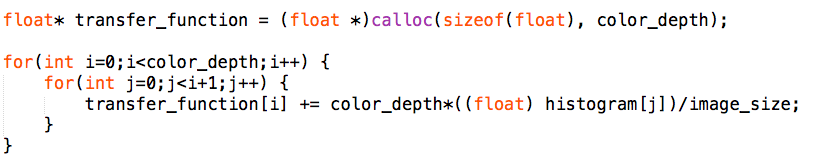
Read the image from bmp file into a one-dimensional character array.



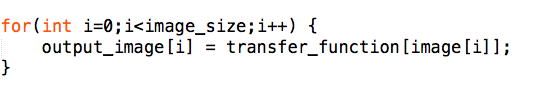
Histogram calculation : For each pixel value calculate its frequency of occurrence and store it in an array - calculation of probability mass function.



Normalize the read values by applying a transfer function - calculate cumulative distribution function of the probability mass function and store the values in another one-dimensional array.



For each pixel in the output image, we store the normalised value for that pixel from the input image.



Write the pixel values in the output bmp file.



**Time Complexity :** For each step, that is, reading from the input file, calculating the probability mass function, normalizing using the transfer function and writing to the output bmp file, involves computation on each pixel value and hence the running time of the algorithm is O(n) where n is the number of pixels.

**Scope of parallelism :**

There are some parts of the code that are inherently serial, for example, reading from and writing to a bmp image needs to be performed serially. This introduces a bottleneck in the performance of the algorithm. However, the portion of the code that deals with the computation of new pixel values from the old pixel values by probability mass function and cumulative distributive function can be parallelised.

Approximately, 2-3% of the code is inherently serial which means that 97-98% of the code can be executed concurrently as obtained by measuring for different image sizes.

**Sections of code that can be parallelised :**

The transfer function value is calculated for each pixel the value of which is independent per pixel. Hence we can parallelize this part of the code by dividing the work as equally as possible among the threads.

Also, it can be seen that for each pixel, the value of the histogram array for all the smaller indices at every iteration for each pixel. Hence, this can be optimise and can be done parallely.

**Effect of increasing problem size on serial code :**

As the problem size increases, the computation increases. Hence, the time taken also increases. It can also be seen that the computations that can be done in parallel without any conflicts increase. This can be considered when we parallelise the code.

**Gustafson - Barsis’ Law** can be applied in this case to understand the trend in speedup for parallel computation using fixed number of processors but increasing the problem size.

The law states that given a parallel program of size using processors, let be the fraction of total execution time spent in serial code. The maximum speedup achieved by the program is

Screen Shot 2015-11-01 at 10.09.46 pm.png

This gives the maximum possible theoretical speedup. Assuming that s = , we have :

* for p = 2 : Ψ 2 + (1-2) \* 0.02 = 1.98
* for p = 4 : Ψ 4 + (1-4) \* 0.02 = 3.96
* for p = 8 : Ψ 8 + (1-8) \* 0.02 = 7.86

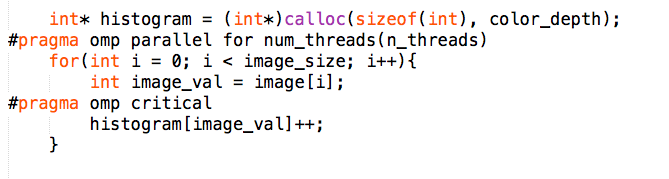
**Section 3 : Description of the Parallel Algorithm**

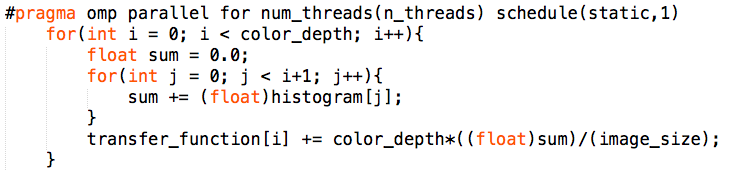
**Pseudocode :**

Read the image from bmp file into a one-dimensional character array.

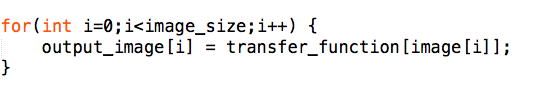


Histogram calculation : For each pixel value calculate its frequency of occurrence and store it in an array - calculation of probability mass function. The assignment of pixel values to the one-dimensional array image[] (which represents the input image) can be parallelised since the threads are accessing elements at different indices. However, since the histogram[] array is a shared variable which all the threads are modifying in each iteration of the loop, one needs to ensure race conditions do not occur. Hence, that part of the code is put in the critical section.



Normalize the read values by applying a transfer function - calculate cumulative distribution function of the probability mass function and store the values in another one-dimensional array. Similar to the above step, we can calculate transfer function in a parallel fashion because each thread operates on a different set of values. Scheduling is used to for achieving almost equal work to all threads.

For each pixel in the output image, we store the normalised value for that pixel from the input image.



Write the pixel values in the output bmp file.



**Time Complexity :**

For each step, that is, reading from the input file, calculating the probability mass function, normalizing using the transfer function and writing to the output bmp file, involves computation on each pixel value and hence the running time of the algorithm is O(n) where n is the number of pixels. The steps that are parallelised run in O(n)/p time where p is the number of pixels but the reading and writing steps still take O(n) time.

**Strategy of parallelization :**

Looking at the serial implementation of the algorithm, it can be seen that reading from and writing to a bmp file needs to be done serially. Even if we attempt to do it in parallel, there would not be much difference in terms of performance since overhead due to false sharing and cache coherency would increase as well.

However, there are two loops - one for calculating probability mass function and the other for calculating the cumulative distribution function which do not have any loop-carried or data dependency and only involve calculating new values from independent pixel values of the input image. Hence, we apply different optimisation strategies to achieve this.

Scheduling can also be used to divide work among the threads in such a way that there is minimum overhead due to false sharing among the threads and work distribution is as uniform as possible.

We have analysed the performance for static and dynamic scheduling and varied chunk size to find the most optimum optimisation strategy.

**Section 4 : Observations and Inferences**

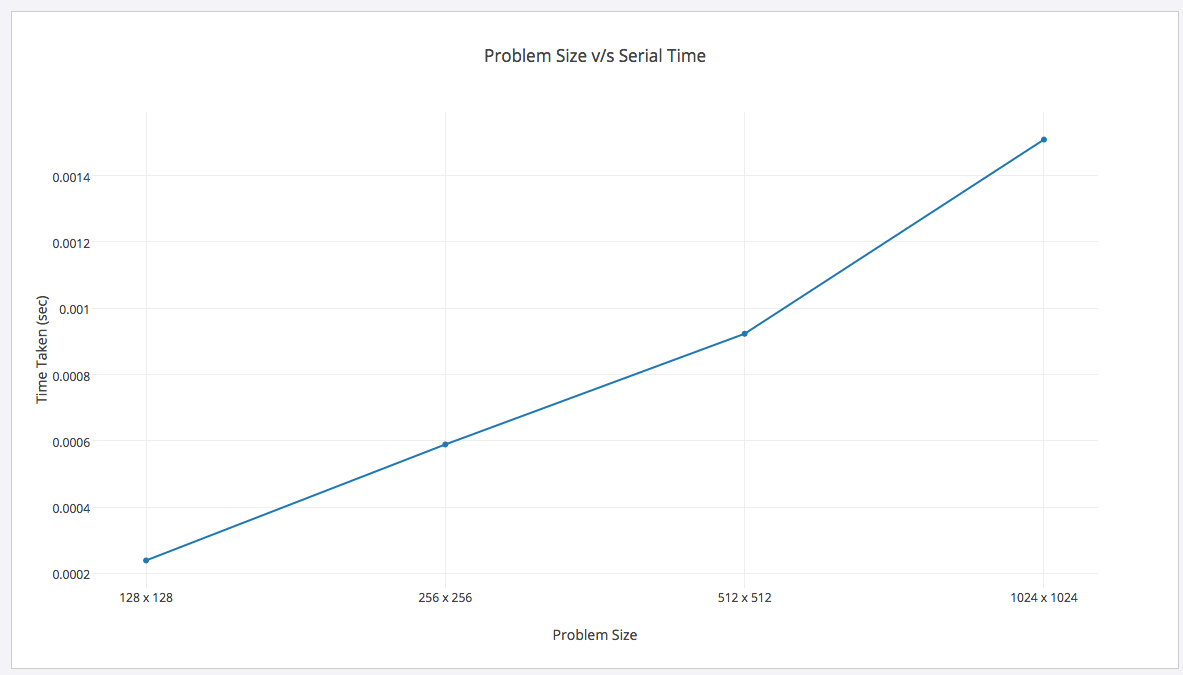
**Results and Discussion :**

For each input image, we have compared the performance of the program with number of threads equal to 2, 4 and 8 threads. Value for higher number of threads have not been mentioned since the performance was lesser as compared to the results with lesser number of threads.

We first compared the performance of static scheduling by varying the chunk size from 2, 4 to 8. As can be seen, the best result is obtained for chunk size = 8. Then we ran the same program with dynamic scheduling and chunk size = 8.

We have computed the values for different image sizes but only included the best results here. The output have been tabulated below:

|  |  |
| --- | --- |
| **Image Dimensions** | **Time taken for serial program** |
| 128 x 128 | 0.000239 |
| 256 x 256 | 0.000589 |
| 512 x 512 | 0.000923 |
| 1024 x 1024 | 0.001509 |
| 2048 x 2048 | 0.000373 |



**Time taken for different versions of the parallel program :**

**Number of threads = 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Image Dimensions** | **Static scheduling** | | | **Dynamic scheduling**  **chunk size = 8** |
| **chunk size = 1** | **chunk size = 4** | **chunk size = 8** |
| 512 x 512 | 0.000257 | 0.000238 | 0.000275 | 0.000339 |
| 1024 x 1024 | 0.000239 | 0.000257 | 0.000233 | 0.000323 |
| 2048 x 2048 | 0.000271 | 0.000253 | 0.000345 | 0.000560 |

**Number of threads = 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Image Dimensions** | **Static scheduling** | | | **Dynamic scheduling**  **chunk size = 8** |
| **chunk size = 1** | **chunk size = 4** | **chunk size = 8** |
| 512 x 512 | 0.000251 | 0.00025 | 0.000243 | 0.000255 |
| 1024 x 1024 | 0.000415 | 0.000401 | 0.000393 | 0.000405 |
| 2048 x 2048 | 0.000504 | 0.000481 | 0.000466 | 0.000471 |

**Number of threads = 8**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Image Dimensions** | **Static scheduling** | | | **Dynamic scheduling**  **chunk size = 8** |
| **chunk size = 1** | **chunk size = 4** | **chunk size = 8** |
| 512 x 512 | 0.000254 | 0.000324 | 0.000330 | 0.000350 |
| 1024 x 1024 | 0.000295 | 0.000369 | 0.000259 | 0.000378 |
| 2048 x 2048 | 0.286524 | 0.000281 | 0.000316 | 0.000398 |

**Speedup for different versions of parallel program :**

**Number of threads = 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Image Dimensions** | **Static scheduling** | | | **Dynamic scheduling**  **chunk size = 8** |
| **chunk size = 1** | **chunk size = 4** | **chunk size = 8** |
| 512 x 512 | 1.404870624 | 1.715613383 | 1.943157895 | 1.712430427 |
| 1024 x 1024 | 1.607028754 | 1.760793466 | 1.902900378 | 1.922292994 |
| 2048 x 2048 | 1.80733945 | 1.860440714 | 1.937704918 | 1.922993492 |

**Number of threads = 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Image Dimensions** | **Static scheduling** | | | **Dynamic scheduling**  **chunk size = 8** |
| **chunk size = 1** | **chunk size = 4** | **chunk size = 8** |
| 512 x 512 | 3.677290837 | 3.692 | 3.798353909 | 3.619607843 |
| 1024 x 1024 | 2.799628942 | 3.301969365 | 3.839694656 | 3.725925926 |
| 2048 x 2048 | 3.517857143 | 3.686070686 | 3.80472103 | 3.76433121 |

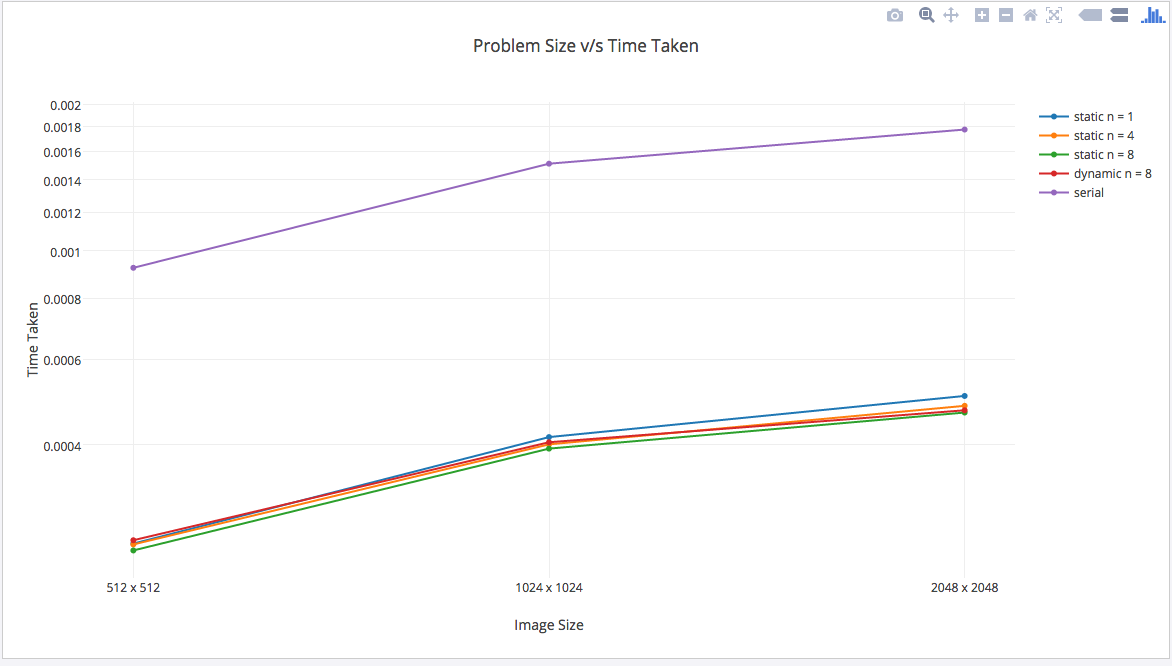
**Number of threads = 8**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Image Dimensions** | **Static scheduling** | | | **Dynamic scheduling**  **chunk size = 8** |
| **chunk size = 1** | **chunk size = 4** | **chunk size = 8** |
| 512 x 512 | 4.120535714 | 4.524509804 | 4.661616162 | 4.458937198 |
| 1024 x 1024 | 5.115254237 | 5.221453287 | 5.826254826 | 5.630597015 |
| 2048 x 2048 | 5.19941349 | 5.832236842 | 5.94966443 | 5.593059937 |

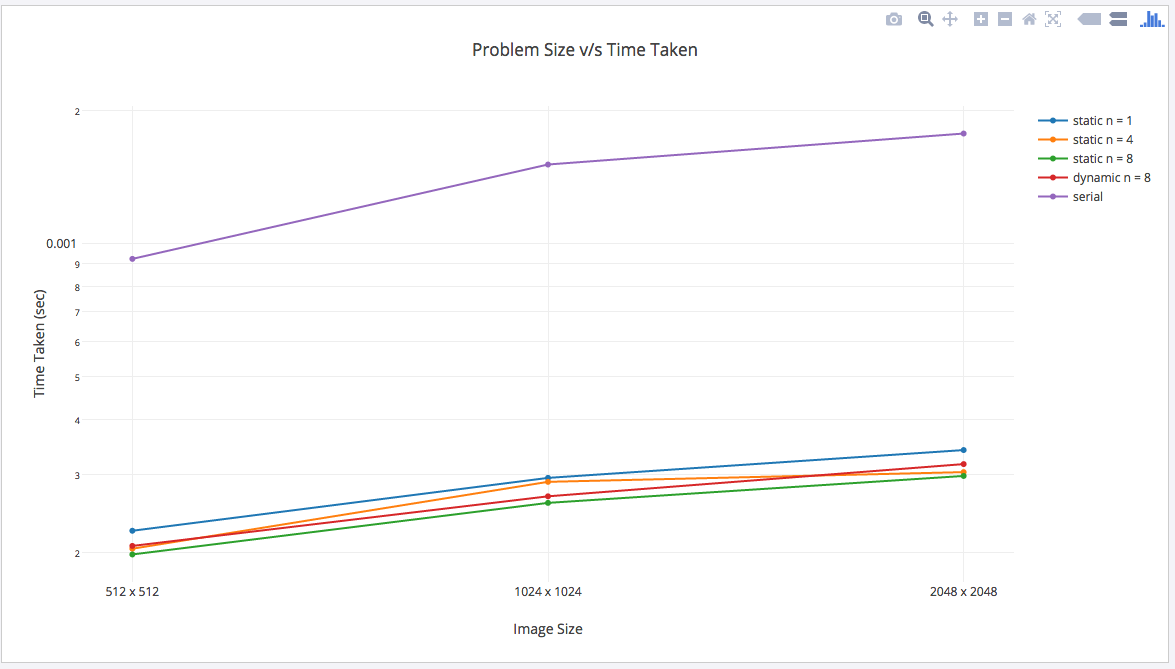
**Time taken as a function of problem size (for different number of threads):**

**Number of threads = 2**



**Number of threads = 4**

**Number of threads = 8**



**Speed-up curve (as a function of no. of processors) :**

**Number of threads = 2**



**Number of threads = 4**

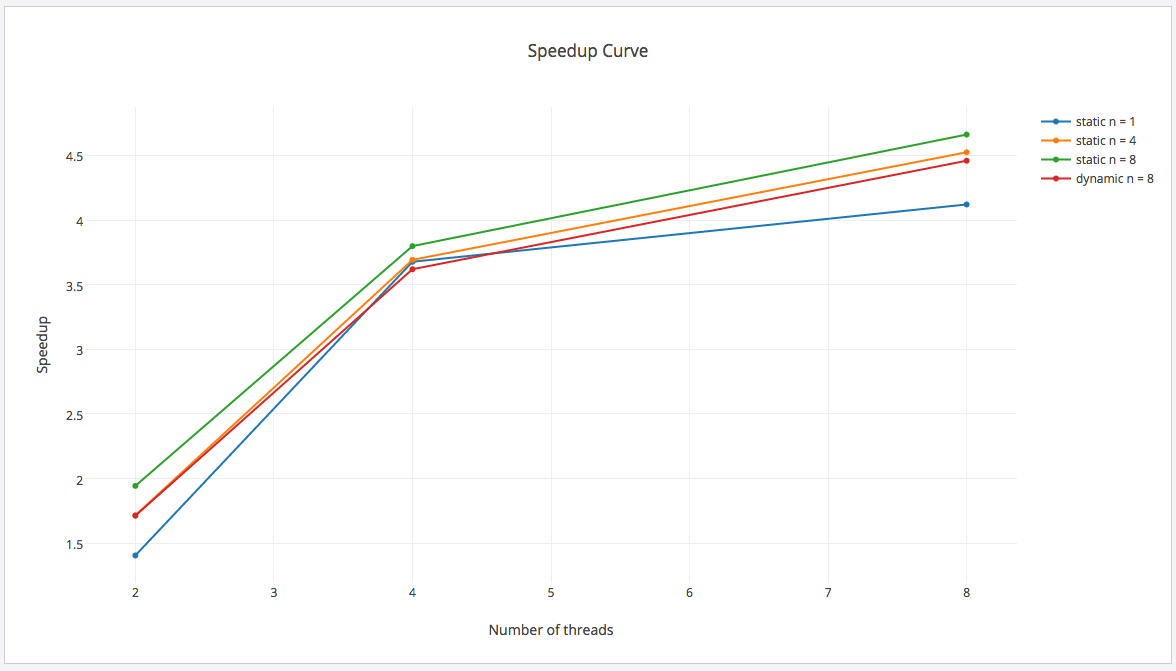


**Number of threads = 8**

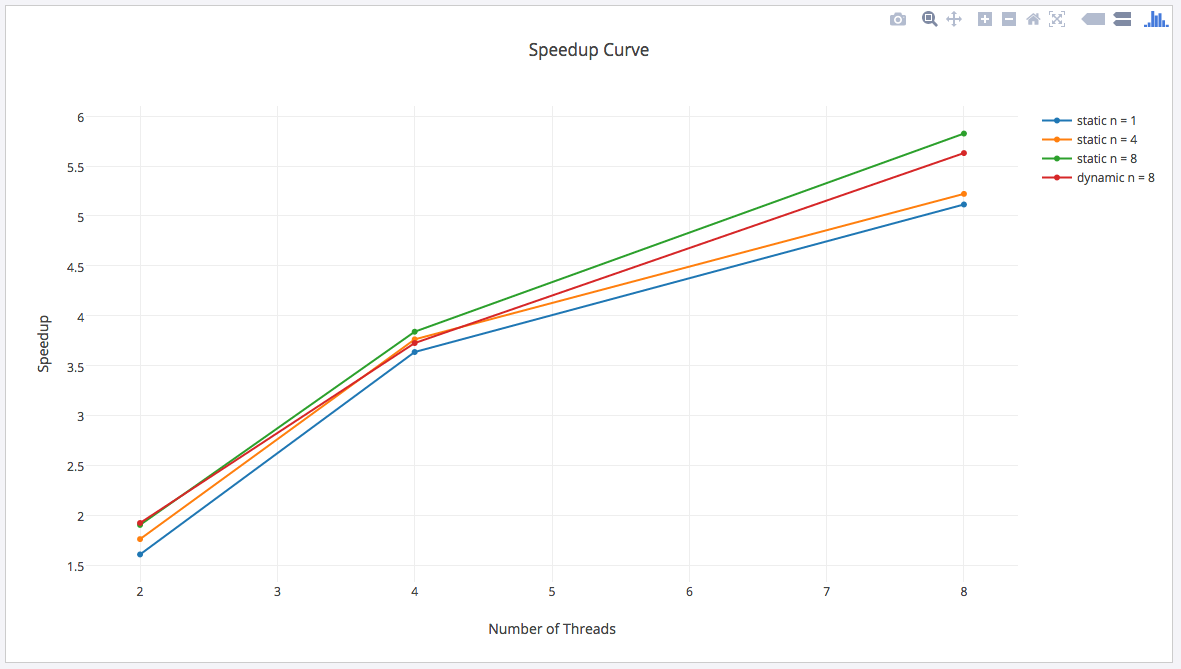


**Speed up as a function of problem size :**

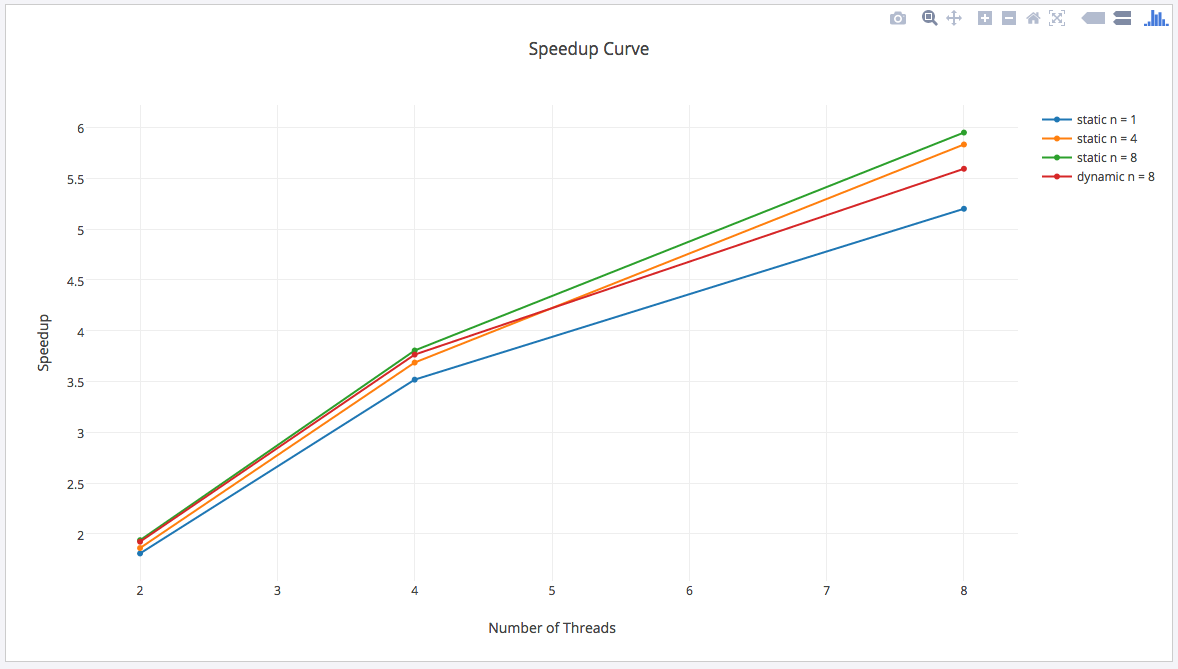
**Image Size : 512 x 512**



**Image Size : 1024 x 1024**



**Image Size : 2048 x 2048**



**How the % of serial part changes with increase in problem size :**

* + As the problem size increases, we see that the portion of serial part also increases. This is the portion of code that involves reading to and writing from a bmp image.
  + As the number of pixels increase, the serial part increases but with respect to the whole code this portion is very much negligible.
  + It does not affect much to the whole implementation of the code.

Considering static scheduling, we have by the **Karp-Flatt Metric** :

**Image Size : 512 x 512**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of threads (p)** | **2** | **4** | **8** |
| **Ψ(n,p)** | 1.715613383 | 3.692 | 4.524509804 |
| **e(n,p)** | 0.016576 | 0.027807 | 0.109735 |

**Image Size : 1024 x 1024**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of threads (p)** | **2** | **4** | **8** |
| **Ψ(n,p)** | 1.760793466 | 3.301969365 | 5.221453287 |
| **e(n,p)** | 0.0136363 | 0.070584 | 0.0760223 |

**Image Size : 2048 x 2048**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of threads (p)** | **2** | **4** | **8** |
| **Ψ(n,p)** | 1.860440714 | 3.686070686 | 5.832236842 |
| **e(n,p)** | 0.007526 | 0.0283957 | 0.053098 |

As can be seen from the above values, the value of e is increasing with the number of processors, (e represents the experimentally determined serial fraction of code), hence it can be concluded that the reason for speedup being lesser than the maximum achievable value is not only due to the inherently serial portion of the code but also the increase in parallel overhead; the other reasons can be :

* process startup time
* process synchronization time
* imbalanced workload
* architectural overhead

which also introduce inefficiency in the system.

**Section 5 : Conclusion**

**Major observations and conclusion :**

* + Time Taken by serial implementation is always more than that in parallel implementation. It shows that parallel implementation decreases time complexity. And as the number of increases, the parallel time taken also decreases.
  + The speedup obtained is maximum for static scheduling with chunk size = 8. Here the work done by all the threads is equal, thus static scheduling works the best. The speedup in dynamic scheduling with chunk size = 8 is similar to speedup in static scheduling with chunk size = 4. This is because, dynamic scheduling is done at run-time, thus there is some overhead as compared static scheduling. In our problem the threads have same amount of computations and thus static scheduling works the best and the most optimum chunk size is 8.
  + As the problem size is increasing the speedup is also increasing, but it is not reaching a maximum speedup it should achieve. Maximum speedup achieved is only 6 with 8 threads. Upto 4 threads the speedup is almost equal to the number of threads which shows that the serial fraction of code is not much significant. After 4 threads, as the number of threads are increasing the speedup is not achieved as expected. This is because after 4 threads the parallel overhead becomes more significant and it increases. Thus speedup is not achieved as expected.

**Scope of further improvement :**

Scalability: With increase in the image size, the computations and the parallel overhead increases. In our program we are making histogram arrays which are of same size as

image size. Thus, a large amount of memory is used in order to execute the program.

In shared memory system, there might be the memory constraints and thus the image size cannot be increased after a certain point. This problem can be solved using MPI. Using MPI would provides scalability to our problem.

The histogram array can be stored in different memories of different processors. As the computations are independent of each other in our program, these computations can be done independently on the processor. The communication between the processors in the program is needed only during reading and writing the image.

**Section 6 : Appendix**

**A comparison of the input and output images and their respective histograms :**

|  |  |
| --- | --- |
| **Image Size : 128x128** | |
| **Input Image** | |
| 128.jpg | 128_histogram.gif |
| **Output of serial program** | |
|  | ou_128_histogram.gif |
| **Output of parallel program** | |
|  | ou_128_histogram.gif |

|  |  |
| --- | --- |
| **Image Size : 256x256** | |
| **Input Image** | |
| 256.jpg | 256_histogram.gif |
| **Output of serial program** | |
| 256_out.jpg | 256_serial_out_histogram.gif |
| **Output of parallel program** | |
| 256_out.jpg | 256_serial_out_histogram.gif |

|  |  |
| --- | --- |
| **Image Size : 512x512** | |
| **Input Image** | |
|  | 512_histogram.gif |
| **Output of serial program** | |
|  | 512_4_out_histogram.gif |
| **Output of parallel program** | |
|  | 512_4_out_histogram.gif |

|  |  |
| --- | --- |
| **Image Size : 1024x1024** | |
| **Input Image** | |
|  | 1024_histogram.gif |
| **Output of serial program** | |
|  | 1024out_histogram.gif |
| **Output of parallel program** | |
|  | 1024out_histogram.gif |

|  |  |
| --- | --- |
| **Image Size : 2048 x 2048** | |
| **Input Image** | |
|  | 2048_histogram.gif |
| **Output of serial program** | |
|  | 2048_out_histogram.gif |
| **Output of parallel program** | |
|  | 2048_out_histogram.gif |