PARALLEL IMPLEMENTATION OF MEDIAN FILTERING

PROJECT REPORT

Name: Prakhar Vipin Jain
Course: MATH 424
Department: CPRE
Net-ID: pvjain05@iastate.edu

INTRODUCTION

Image filters can be classified as linear or nonlinear. Linear filters are also known as convolution filters as they can be represented using a matrix multiplication. Thresholding and image equalization are examples of nonlinear operations, as is the median filter. Median filtering is a nonlinear operation often used in image processing to reduce "salt and pepper" noise. A median filter is more effective than convolution when the goal is to simultaneously reduce noise and preserve the edges in an image.

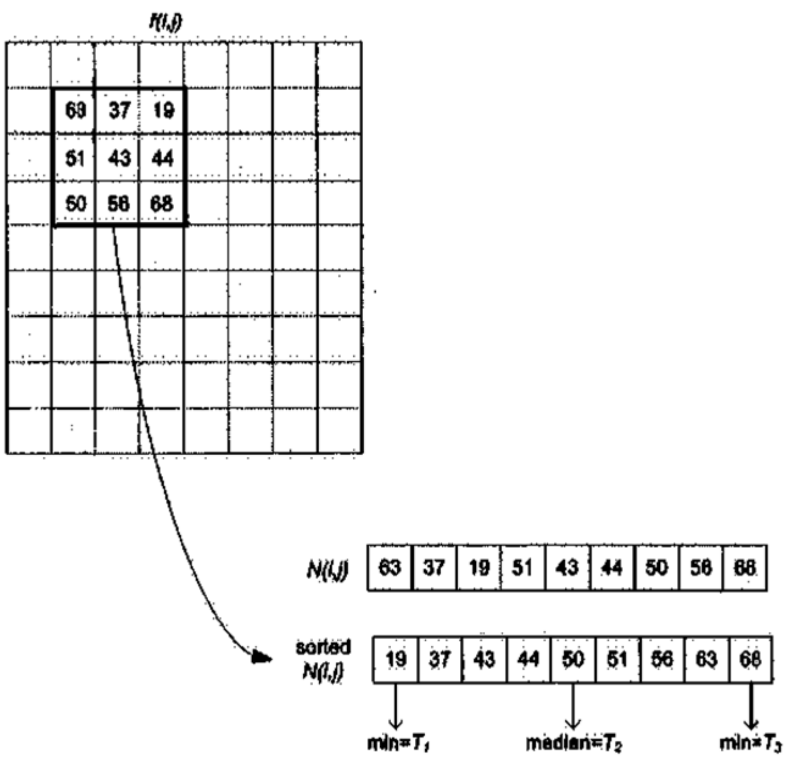
In median filtering, the noisy pixel considered is replaced by the median value of the neighboring pixels. The number of neighboring pixels depend on the size of the mask. Typically, a (3×3) mask is the smallest mask that can be used. Other than this, mask sizes of (5×5), (7×7), (9×9) and so on can also be used. Thus, the median filter works by sliding through the image pixel by pixel and replacing them by the median value obtained among the neighboring pixels. The median is calculated by first sorting the values of all the pixels in the increasing order of magnitude and then picking up the middle value from this sorted array of pixel values. This is one of the reasons why the size of the mask is always an odd multiple.

An image consists of millions of pixels and better the quality of the image, more the number of pixels. Processing each and every pixel is a time-consuming task and if done sequentially, i.e. for every mask, sorting the pixel values and then replacing the considered pixel with the median value ultimately results in slow processing of the image. This time is inversely proportional to the size of the mask. This means that as the size of the mask decreases, the processing time increases.

Comparatively, parallelizing the processing task can help reduce the processing time of the image considerably. The serial program for the above implementation of the median filter has been constructed and is provided later in this report. Looking ahead, I will parallelize the deduced serial code in OpenMP. This will include dividing the iterations pertaining to the sorting the pixel values and finding the median among them for every sliding window.

WORKED OUT EXAMPLE

Given below is an example of how Median Filtering is done on an image. Consider the matrix given below as the part of the image. Here a (3 x 3) window is used to filter the given image. For altering the boundary elements of the image, their duplicates are considered at the respective edge. The elements or the pixel values within the window size are sorted and the considered pixel is replaced with the median value.



THE SELECTED OPENMP CLAUSES

```
# pragma omp parallel for collapse(2) ordered num_threads(thread_count)
default(none) shared(image, filteredimage, rows, cols, n, time)
firstprivate(pixel_values) private(my_rank, t3, t4, p, rr, cc)
schedule(static,1)
```

The above schedule clause which is ‘static’ with chunk size 1 is selected because it has the best performance among all the other scheduling techniques which included dynamic, guided, auto and runtime. Along with running the program with different schedules, I also tried different schedules with different chunk sizes, viz. 1, 4, 8, 16 and default chunk sizes.

For static schedule, the default value of the chunk size is the number of iterations divided by the number of threads i.e. $\frac{Number\ of\ iterations}{Number\ of\ \ threads}$. For dynamic and guided schedules, the default value for the chunk size is 1. As for auto and runtime schedules, there is no chunk size. The compiler allocates the threads at runtime.

As the section of the code that I have parallelized contains ‘for’ loops, I have used ‘parallel for’ construct for parallelization. I have used an additional collapse clause which I will mention later. Another construct I have used is the ‘ordered’ construct which is used to sequentialize and orders the execution of ordered regions. Unordered populating of the rows and columns with the filtered pixel values would have distorted the image. Thus, I have used the ‘ordered’ construct while populating the rows and columns of the output image.

The scope of the variables was also defined as for parallelization, some variables needed to be shared, private and firstprivate. The variables that were shared are {image, filteredimage, rows, cols, n, time}. The variable ‘image’ is a 2D vector which contains the input image. As each thread is only reading the image, there is no problem of race conditions between the threads. The ‘filteredimage’ variable is a pointer to a 2D vector and points to the output image. It is shared amonf the thread because it is the output image and every thread has to populate the rows and columns of it by the pixels they have computed individually. The variables, ‘rows’ and ‘columns’ contains the number of rows and columns of the input image passed and ‘n’ is the size of the window/mask that performs the median filtering over (n x n) area in the image. These variables are also shared among the threads because the threads only have to read them and so there is no problem of race conditions. The last shared variable is 'time' which is an array used to store the time required by each thread to compute their own chunk. As each thread stores their time in it based on their rank, this variable is also shared.

The variables that were private to each thread are {pixel_values, my_rank, t3, t4, p, rr, cc}. The variable 'pixel_values' is firstprivate because it needed to be initialized with the value that it encounters in the previous construct. If it would be private, every time the thread returned after an iteration, the value of pixel_values would be zero which we do not want. Other variables like 'my_rank', 't3' and 't4' relate to an individual thread which is reason for them to be private. Variables 'p', 'rr' and 'cc' are used by each thread to compute their own chunk and so are also private.

OPENMP NEW CONCEPT USED

The collapse clause may be used to specify how many loops are associated with the loop construct. The parameter of the collapse clause must be a constant positive integer expression. If a collapse clause is specified with a parameter value greater than 1, then the iterations of the associated loops to which the clause applies are collapsed into one larger iteration space that is then divided according to the schedule clause. The sequential execution of the iterations in these associated loops determines the order of the iterations in the collapsed iteration space. If no collapse clause is present or its parameter is, the only loop that is associated with the loop construct for the purposes of determining how the iteration space is divided according to the schedule clause is the one that immediately follows the loop directive.

I have mentioned before that I have used another clause named 'collapse' which is used to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread. It also increases the scalability of the program. In this program, I have collapsed two for loops (i and j) thereby increasing the iteration space for rows and columns of the image.

RESULTS

The input and output images for both serial and parallel execution are of .pgm format which is a Portable Grayscale Map. A PGM image represents a grayscale graphic image. There are many pseudo-PGM formats in use where everything is as specified herein except for the meaning of individual pixel values. For most purposes, a PGM image can just be thought of an array of arbitrary integers, and all the programs in the world that think they're processing a grayscale image can easily be tricked into processing something else.

*The results displayed below are for Input_Image3.pgm

For serial execution –

Input Image:

Output Image:

This above filtering is done by using window size = 9. One can notice the stars to be blur in the output image which indicates reduction of noise from the image.
Execution time = 263.193462 secs

For parallel execution –

Input Image: Output Image:

This is the output image with static scheduling with chunk size 1 which proved to be the optimum schedule for parallelization.

Execution time = 92.773567 secs

Verification with MATLAB-

DATA ANALYSIS

*<u>NOTE- All timings are in seconds</u>

<u>Data Set 1</u>: Input_Image1.pgm (18,994 KB)

Threads →	**Chunk size ↓**	1	2	4	8	16
Schedules:	1	159.845610	117.618669	87.075414	73.088391	59.778781
	4	159.741286	132.329936	127.275453	147.192496	156.108165
Static	8	159.994710	147.171165	144.139437	163.814115	171.852579
	16	160.001472	153.559104	153.040254	171.760347	205.658922
Dynamic	1	159.870233	115.997256	87.090787	72.831194	60.253612
	4	160.067111	132.131750	127.503150	149.048226	152.275864
	8	160.124409	146.360010	145.586016	162.689667	170.716539
	16	160.017528	153.391619	153.119732	172.553518	203.815412
Guided	1	160.182147	160.649851	160.002478	160.137532	162.481793
	4	160.065258	161.689344	160.550149	160.199708	160.618009
	8	160.112786	161.259638	160.944127	161.257453	160.583052
	16	159.954120	160.285715	160.129530	160.128016	160.743391
Auto	No chunk size	159.585215	160.070749	160.068013	161.014960	160.257501
Runtime		159.844471	103.295304	75.054813	74.117641	60.477173

- The serial time = 160.820007 secs
- Best case Parallel execution time = 59.778781 secs
- The workload distribution for static schedule was even and it is clear from the execution times of each thread out of 16 threads given below:

  Time taken by thread 0 to compute its chunk = 59.175427 secs
  Time taken by thread 1 to compute its chunk = 59.308516 secs
  Time taken by thread 2 to compute its chunk = 59.085974 secs
  Time taken by thread 3 to compute its chunk = 59.169525 secs
  Time taken by thread 4 to compute its chunk = 59.005827 secs
  Time taken by thread 5 to compute its chunk = 59.196315 secs
  Time taken by thread 6 to compute its chunk = 59.231253 secs
  Time taken by thread 7 to compute its chunk = 59.206661 secs
  Time taken by thread 8 to compute its chunk = 59.226837 secs
  Time taken by thread 9 to compute its chunk = 59.045139 secs
  Time taken by thread 10 to compute its chunk = 58.899492 secs
  Time taken by thread 11 to compute its chunk = 59.053454 secs
  Time taken by thread 12 to compute its chunk = 58.921677 secs

Time taken by thread 13 to compute its chunk = 58.893360 secs
Time taken by thread 14 to compute its chunk = 58.924835 secs
Time taken by thread 15 to compute its chunk = 58.895725 secs

- The auto and guided schedule show similar timings that is fairly close to the serial execution time. This may be because of the fact that guided schedule allots a chunk of data to the threads only when the thread is done computing its previous chunk and asks for a new one. But as the parallel code contains a critical as well as an ordered clause, there is lot of overhead as threads have to wait for a long time along with their chunks to be executed.
  The workload distribution was uneven for the 'auto' schedule:
  Time taken by thread 0 to compute its chunk = 10.605937 secs
  Time taken by thread 1 to compute its chunk = 21.073863 secs
  Time taken by thread 2 to compute its chunk = 31.768825 secs
  Time taken by thread 3 to compute its chunk = 42.804017 secs
  Time taken by thread 4 to compute its chunk = 54.107703 secs
  Time taken by thread 5 to compute its chunk = 65.698848 secs
  Time taken by thread 6 to compute its chunk = 77.231345 secs
  Time taken by thread 7 to compute its chunk = 88.536510 secs
  Time taken by thread 8 to compute its chunk = 100.280961 secs
  Time taken by thread 9 to compute its chunk = 111.037032 secs
  Time taken by thread 10 to compute its chunk = 120.036634 secs
  Time taken by thread 11 to compute its chunk = 128.396288 secs
  Time taken by thread 12 to compute its chunk = 135.420295 secs
  Time taken by thread 13 to compute its chunk = 143.372302 secs
  Time taken by thread 14 to compute its chunk = 152.289238 secs
  Time taken by thread 15 to compute its chunk = 160.213678 secs
- **This uneven distribution was <u>true</u> for 'auto' schedule for all the other data sets as well.**
- I tried running the best schedule i.e. static with chunk size 1 with "numactl --membind=0,1 ./median_parallel.sh". The execution time was **<u>59.006019</u>** secs which is further less than 59.778781 secs.
- As the execution time for parallel execution started increasing from the 16th thread for chunk size 1, I did not calculate for higher number of threads.

<u>Data Set 2</u>: **Input_Image2.pgm (23,438 KB)**

Threads →	**Chunk size ↓**	1	2	4	8	16
Schedules:	1	202.868598	155.673198	107.702861	97.195866	74.463772
Static	4	202.218307	167.885854	163.828961	182.969107	195.442512
	8	202.468531	185.484520	176.545186	198.018291	209.739152
	16	202.451368	189.788451	189.017553	210.545218	238.048461
Dynamic	1	202.054862	149.843054	108.030969	93.826356	75.680126
	4	202.424351	136.485183	119.554039	143.704134	161.065130
	8	201.878542	145.184465	144.845159	198.516521	210.884027
	16	201.715380	153.498456	153.197602	209.148539	228.541498
Guided	1	201.985451	202.763191	202.193746	202.784561	202.681002
	4	202.775829	202.661943	202.200147	201.624889	202.018359
	8	202.488960	201.980238	202.605571	201.129203	201.997420
	16	203.008487	202.484611	201.958412	201.731948	201.488351
Auto	No chunk size	202.487206	201.991024	202.624801	202.485100	202.548015
Runtime		202.668514	123.947124	111.808369	93.687449	76.087816

- The serial time = 204.369995 secs
- Best case Parallel execution time = 74.463772 secs
- The workload distribution for static schedule was even and it is clear from the execution times of each thread out of 16 threads given below:

  Time taken by thread 0 to compute its chunk = 74.989118 secs
  Time taken by thread 1 to compute its chunk = 74.854582 secs
  Time taken by thread 2 to compute its chunk = 74.757406 secs
  Time taken by thread 3 to compute its chunk = 74.926200 secs
  Time taken by thread 4 to compute its chunk = 74.896859 secs
  Time taken by thread 5 to compute its chunk = 74.953371 secs
  Time taken by thread 6 to compute its chunk = 75.080892 secs
  Time taken by thread 7 to compute its chunk = 75.020143 secs
  Time taken by thread 8 to compute its chunk = 74.917979 secs
  Time taken by thread 9 to compute its chunk = 74.909147 secs
  Time taken by thread 10 to compute its chunk = 74.639726 secs
  Time taken by thread 11 to compute its chunk = 74.650440 secs
  Time taken by thread 12 to compute its chunk = 74.752266 secs
  Time taken by thread 13 to compute its chunk = 74.680941 secs
  Time taken by thread 14 to compute its chunk = 74.678205 secs
  Time taken by thread 15 to compute its chunk = 74.673349 secs

- I tried running the best schedule i.e. static with chunk size 1 with “numactl --membind=0,1 ./median_parallel.sh”. The execution time was **74.219392** secs which is further less than 74.463772 secs.

Data Set 3: **Input_Image3.pgm (27,547 KB)**

Threads →	**Chunk size ↓**	1	2	4	8	16
Schedules:	1	263.193462	173.573315	135.101077	117.154057	92.773567
	2	262.936057	173.650104	148.154255	168.441251	178.032077
Static	4	263.778588	215.011170	208.020343	236.099375	249.875132
	8	263.395564	239.524179	236.395383	268.244220	280.195565
	16	263.786110	251.829353	251.352660	282.789761	338.788450
	Default	263.392294	263.720043	263.095824	263.491843	263.631292
Dynamic	1	264.383264	155.191040	137.229584	114.507565	93.079510
	2	264.421474	163.394685	151.830979	164.835083	171.337266
	4	263.472723	211.308588	209.950337	236.196474	248.357794
	8	263.684505	240.456126	236.932889	267.279579	278.945360
	16	263.006030	251.670132	251.147524	281.705669	336.945297
Guided	1	263.351812	263.725511	262.886220	263.437402	263.538888
	2	262.935334	263.807951	263.845951	262.584845	263.446711
	4	263.865233	263.771412	262.555214	263.110249	262.473829
	8	262.478510	263.018544	263.885268	263.257841	263.985177
	16	262.110218	263.784147	263.101128	263.124781	262.577452
Auto	No chunk	261.665263	263.798283	262.984063	263.504336	262.478512
Runtime	size	264.556103	209.147851	176.001476	120.785421	94.682036

- The serial time = 263.193462 secs
- Best case Parallel execution time = 92.773567 secs
- The workload distribution for static schedule was even and it is clear from the execution times of each thread out of 16 threads given below:

  Time taken by thread 0 to compute its chunk = 92.373467 secs
  Time taken by thread 1 to compute its chunk = 92.574180 secs
  Time taken by thread 2 to compute its chunk = 92.641522 secs
  Time taken by thread 3 to compute its chunk = 92.522662 secs
  Time taken by thread 4 to compute its chunk = 92.512363 secs
  Time taken by thread 5 to compute its chunk = 92.570308 secs
  Time taken by thread 6 to compute its chunk = 92.562178 secs
  Time taken by thread 7 to compute its chunk = 92.545392 secs

Time taken by thread 8 to compute its chunk = 92.533315 secs
Time taken by thread 9 to compute its chunk = 92.658345 secs
Time taken by thread 10 to compute its chunk = 92.488986 secs
Time taken by thread 11 to compute its chunk = 92.150117 secs
Time taken by thread 12 to compute its chunk = 92.200594 secs
Time taken by thread 13 to compute its chunk = 92.132365 secs
Time taken by thread 14 to compute its chunk = 92.206376 secs
Time taken by thread 15 to compute its chunk = 92.359847 secs

- I tried running the best schedule i.e. static with chunk size 1 with "numactl --membind=1 ./median_parallel.sh". The execution time was **91.652204** secs which is further less than 92.773567 secs.

Data Set 4: **Input_Image4.pgm (40,573 KB)**

Threads →	**Chunk size ↓**	1	2	4	8	16
Schedules:	1	320.924301	203.797932	168.335727	154.422265	122.814190
	4	319.014128	265.834311	258.315386	297.454754	306.212876
Static	8	319.814752	293.499979	290.561753	325.366131	341.880183
	16	319.854721	307.311527	315.986562	347.657032	395.676989
Dynamic	1	319.065278	205.469308	180.126757	150.502692	125.166953
	4	319.743651	259.147529	238.875028	294.217590	309.305614
	8	319.558102	267.079641	261.066913	302.548642	323.647100
	16	318.001423	290.914560	289.156977	315.625891	336.078178
Guided	1	320.904215	321.011538	321.001472	320.802146	320.512184
	4	319.421397	319.514489	320.744121	321.014792	321.110457
	8	319.234810	320.021549	319.874842	320.917356	321.014736
	16	318.079124	321.147260	320.904215	319.704681	320.731841
Auto	No chunk size	318.661973	319.029364	319.196331	318.510953	318.759015
Runtime		320.187503				

- The serial time = 322.440002 secs
- Best case Parallel execution time = 122.814190 secs
- The workload distribution for static schedule was even and it is clear from the execution times of each thread out of 16 threads given below:

Time taken by thread 0 to compute its chunk = 122.228728 secs
Time taken by thread 1 to compute its chunk = 121.981067 secs
Time taken by thread 2 to compute its chunk = 122.079668 secs
Time taken by thread 3 to compute its chunk = 122.142349 secs

Time taken by thread 4 to compute its chunk = 122.233528 secs
Time taken by thread 5 to compute its chunk = 122.263985 secs
Time taken by thread 6 to compute its chunk = 122.359473 secs
Time taken by thread 7 to compute its chunk = 122.300504 secs
Time taken by thread 8 to compute its chunk = 122.063045 secs
Time taken by thread 9 to compute its chunk = 122.175282 secs
Time taken by thread 10 to compute its chunk = 121.614302 secs
Time taken by thread 11 to compute its chunk = 121.623304 secs
Time taken by thread 12 to compute its chunk = 121.514588 secs
Time taken by thread 13 to compute its chunk = 121.604645 secs
Time taken by thread 14 to compute its chunk = 121.675647 secs
Time taken by thread 15 to compute its chunk = 121.569140 secs

- I tried running the best schedule i.e. static with chunk size 1 with “numactl --membind=1 ./median_parallel.sh”. The execution time was **121.315165**secs which is further less than 122.814190 secs.

Data Set 5: **Input_Image5.pgm (41,189 KB)**

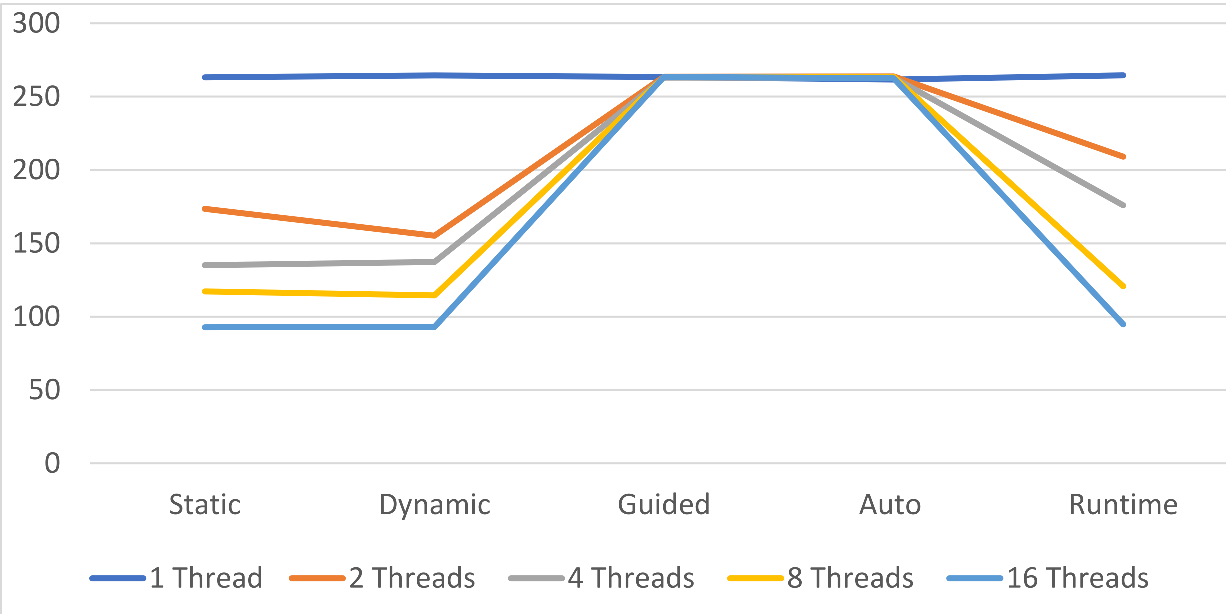
Threads →	**Chunk size ↓**	1	2	4	8	16
Schedules:	1	317.307204	221.980497	171.382066	149.577882	122.572143
	4	316.736039	262.641485	252.715101	292.006935	304.854894
Static	8	316.041560	289.860343	285.962997	321.478609	338.540281
	16	317.114723	306.500580	303.485367	340.555992	399.463339
	Default	318.349357	317.181895	316.568222	316.785780	316.977620
Dynamic	1	318.684126	235.159015	169.650887	149.213248	123.151026
	4	318.741452	261.880632	254.220357	292.812627	300.445066
	8	321.844758	289.923953	286.936467	323.372678	348.505445
	16	317.416015	303.859942	303.520297	352.360554	400.940202
Guided	1	317.236978	316.847520	317.235275	317.500143	317.040865
	4	316.147520	317.199687	316.774920	316.845112	317.107772
	8	316.114759	318.222753	317.964102	316.674102	316.112473
	16	317.521470	316.740023	316.680122	317.657012	318.831931
Auto	No chunk size	317.171288	316.815879	316.605792	317.761608	316.632227
Runtime		316.632227	198.991419	171.962305	147.412915	123.885870

- The serial time = 318.700012 secs
- Best case Parallel execution time = 122.572143 secs

- The workload distribution for static schedule was even and it is clear from the execution times of each thread out of 16 threads given below:
  Time taken by thread 0 to compute its chunk = 121.733547 secs
  Time taken by thread 1 to compute its chunk = 122.140164 secs
  Time taken by thread 2 to compute its chunk = 122.249230 secs
  Time taken by thread 3 to compute its chunk = 121.971068 secs
  Time taken by thread 4 to compute its chunk = 121.959221 secs
  Time taken by thread 5 to compute its chunk = 121.867586 secs
  Time taken by thread 6 to compute its chunk = 122.090898 secs
  Time taken by thread 7 to compute its chunk = 122.162658 secs
  Time taken by thread 8 to compute its chunk = 122.206198 secs
  Time taken by thread 9 to compute its chunk = 122.036070 secs
  Time taken by thread 10 to compute its chunk = 121.479819 secs
  Time taken by thread 11 to compute its chunk = 121.435774 secs
  Time taken by thread 12 to compute its chunk = 121.574673 secs
  Time taken by thread 13 to compute its chunk = 121.449644 secs
  Time taken by thread 14 to compute its chunk = 121.457294 secs
  Time taken by thread 15 to compute its chunk = 121.275871 secs
- I tried running the best schedule i.e. static with chunk size 1 with “numactl --membind=0,1 ./median_parallel.sh”. The execution time was **122.101921** secs which is further less than 122.572143 secs.

STATISTICAL ANALYSIS

Execution time vs Schedules for different number of threads for Input_Image3.pgm-



The above trend is the same for all the 5 output images

Serial Execution Time vs Best-case Parallel Execution Time for all images-

PERFORMANCE ANALYSIS

Data Set 1: Input_Image1.pgm (18,994 KB)

Speedup = $\frac{Tserial}{Tparallel}$

= $\frac{160.820007}{59.778781}$

~ 2.69

Efficiency = $\frac{Speedup}{\text{No.of threads}}$

= $\frac{2.69}{16}$

= 16.8%

Data Set 2: Input_Image2.pgm (23,438 KB)

Speedup = $\frac{Tserial}{Tparallel}$

= $\frac{204.369995}{74.463772}$

~ 2.74

Efficiency = $\frac{Speedup}{\text{No.of threads}}$

= $\frac{2.74}{16}$

= 17.12%

Data Set 3: Input_Image3.pgm (27,547 KB)

Speedup = $\frac{Tserial}{Tparallel}$

= $\frac{263.193462}{92.773567}$

~ 2.83

Efficiency = $\frac{Speedup}{\text{No.of threads}}$

= $\frac{2.83}{16}$

= 17.5%

<u>Data Set 4</u>: Input_Image4.pgm (40,573 KB)

Speedup = $\frac{Tserial}{Tparallel}$

= $\frac{322.440002}{122.814190}$

~ 2.63

Efficiency = $\frac{Speedup}{\text{No.of threads}}$

= $\frac{2.63}{16}$

= 16.43%

<u>Data Set 5</u>: Input_Image5.pgm (41,189 KB)

Speedup = $\frac{Tserial}{Tparallel}$

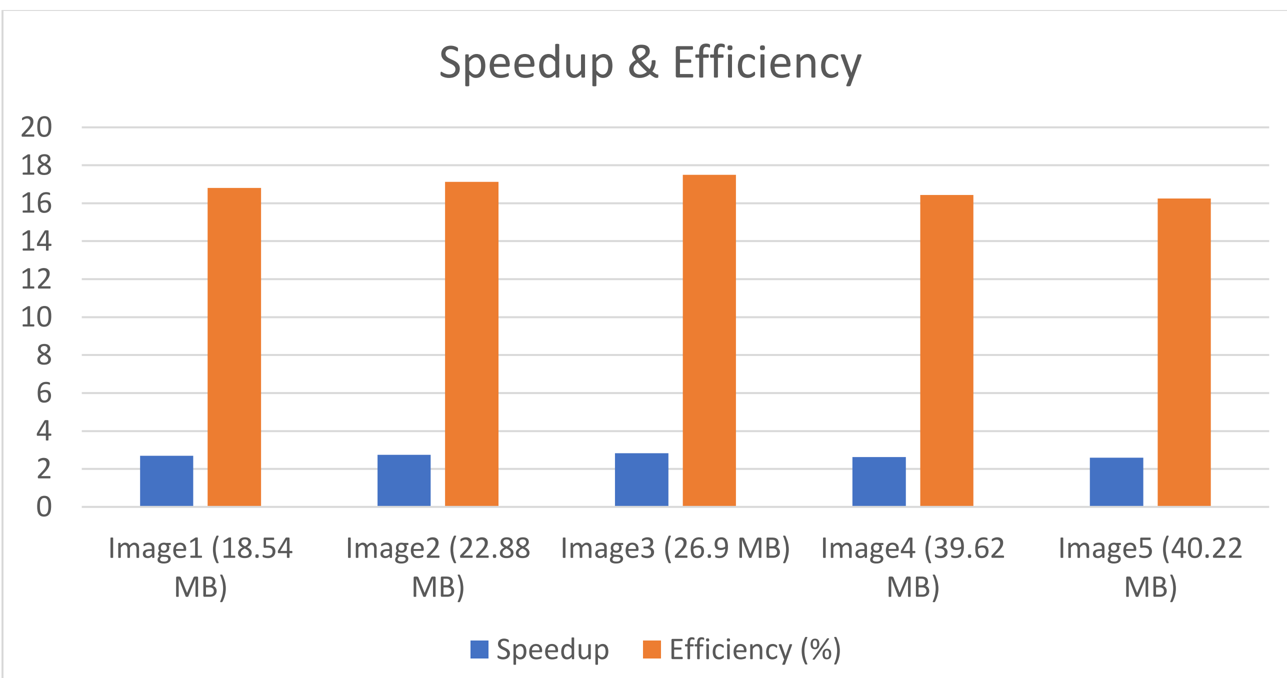
= $\frac{318.700012}{122.572143}$

~ 2.6

Efficiency = $\frac{Speedup}{\text{No.of threads}}$

= $\frac{2.6}{16}$

= 16.25%



Scalability:

- We can see that the efficiency is almost constant for every data set and there is no significant variation as the image size increases.
- That is, the efficiency does not change as the data set/size increases.
- Thus, the parallelized program can be said to be weakly scalable.

APPENDIX

A. Parallel Code:

There are five files included: The main program c file, a function c file, a header file and two shell files for running the serial and parallel programs.

1. median_parallel.c

```
/***************************************************************************
***
* Program: median_parallel.c
* Purpose: This program will apply a median filter to an image with a user
* specified window size.
* Run: ./median_parallel.sh
* Name: Prakhar Jain, MATH 424
****************************************************************************
**/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <omp.h>
#include "imageio.h"

void insertion_sort(unsigned char *item, int count);

int main(int argc, char *argv[])
{
   char *inputfilename1=NULL, *inputfilename2=NULL, *inputfilename3=NULL,
*inputfilename4=NULL, *inputfilename5=NULL;
   char *outputfilename = NULL;
   int n=0, rows, cols, i, c, flag=0;
   unsigned char **image1=NULL, **image2=NULL, **image3=NULL, **image4=NULL,
**image5=NULL, **filteredimage1 = NULL, **filteredimage2 = NULL,
**filteredimage3 = NULL, **filteredimage4 = NULL, **filteredimage5 = NULL;
   int thread_count,t;
   double t1,t2;
   void median_filter(unsigned char **image, int rows, int cols, int n,
unsigned char ***filteredimage, int thread_count);


/***************************************************************************
   * Get the command line parameters.

***************************************************************************/
   for(i=1;i<argc;i++)
   {
```

```
      if(strcmp(argv[i], "-n") == 0)
      {
        n = atoi(argv[i+1]);
        i++;
      }
      else
      {
         if(inputfilename1 == NULL) inputfilename1 = argv[i];
         else if(inputfilename2 == NULL) inputfilename2 = argv[i];
         else if(inputfilename3 == NULL) inputfilename3 = argv[i];
         else if(inputfilename4 == NULL) inputfilename4 = argv[i];
         else if(inputfilename5 == NULL) inputfilename5 = argv[i];
         else if(outputfilename == NULL) outputfilename = argv[i];
         else thread_count = strtol(argv[i], NULL, 10);
      }
   }

   if((n <= 0) || (inputfilename1==NULL) ||  (inputfilename2==NULL) ||
(inputfilename3==NULL) ||  (inputfilename4==NULL) || (inputfilename5==NULL)
||(outputfilename==NULL) || (thread_count <= 0))
   {


printf("\n*******************************************************************
*************\n");
      printf("This program will apply a median filter to an image. You need
to specify the\n");
      printf("size of the window to use in median filtering the image (i.e. -
n 5), \n");
      printf("the 5-input images to process, the name of an output file\n");
      printf("in which the median filtered image will be written and the
number of threads.\n");
      printf("Thus, you could run the program as follows:\n");
      printf("\n");
      printf("Ex: median -n 9 Input_Image1.pgm Input_Image2.pgm
Input_Image3.pgm Input_Image4.pgm Input_Image5.pgm medianfiltered.pgm 8\n");
      printf("\n");

printf("*********************************************************************
***********\n");

      fprintf(stderr, "\n<USAGE> median -n # inputPGMfile outputPGMfile
#of_threads\n\n");
      exit(1);
   }


/****************************************************************************
   * printf("Inputfilename = %s\n", inputfilename);
   * printf("Outputfilename = %s\n", outputfilename);
```

```
   * printf("n = %d\n", n);
****************************************************************************/


/****************************************************************************
   * Read in the PGM image from the file.

****************************************************************************/

 printf("For Input_Image1.pgm - 1\n");
 printf("For Input_Image2.pgm - 2\n");
 printf("For Input_Image3.pgm - 3\n");
 printf("For Input_Image4.pgm - 4\n");
 printf("For Input_Image5.pgm - 5\n");
 printf("Choose which image you want to smooth using median filter : ");
 scanf("%d", &c);
 printf("\n");

 switch(c)
   {
     case 1:
      {
        if(read_pgm_image(inputfilename1, &image1, &rows, &cols) == 0)
exit(1);
        flag=1;
      }
      break;
     case 2:
      {
        if(read_pgm_image(inputfilename2, &image2, &rows, &cols) == 0)
exit(1);
        flag=2;
      }
      break;
     case 3:
      {
        if(read_pgm_image(inputfilename3, &image3, &rows, &cols) == 0)
exit(1);
        flag=3;
      }
      break;
     case 4:
      {
        if(read_pgm_image(inputfilename4, &image4, &rows, &cols) == 0)
exit(1);
        flag=4;
      }
      break;
     case 5:
      {
```

```
        if(read_pgm_image(inputfilename5, &image5, &rows, &cols) == 0)
exit(1);
        flag=5;
      }
      break;
   }



/***************************************************************************
   * Median filter the image.

***************************************************************************/
  if(flag==1)
  {
    t1= omp_get_wtime();
      median_filter(image1, rows, cols, n, &filteredimage1, thread_count);
    t2= omp_get_wtime();
  }
  if(flag==2)
  {
    t1= omp_get_wtime();
      median_filter(image2, rows, cols, n, &filteredimage2, thread_count);
    t2= omp_get_wtime();
  }
  if(flag==3)
  {
    t1= omp_get_wtime();
      median_filter(image3, rows, cols, n, &filteredimage3, thread_count);
    t2= omp_get_wtime();
  }
  if(flag==4)
  {
    t1= omp_get_wtime();
      median_filter(image4, rows, cols, n, &filteredimage4, thread_count);
    t2= omp_get_wtime();
  }
  if(flag==5)
  {
    t1= omp_get_wtime();
      median_filter(image5, rows, cols, n, &filteredimage5, thread_count);
    t2= omp_get_wtime();
  }


/***************************************************************************
   * Print out the size of window used to compute the median
   * and the time taken to do the median filtering.
```

```
**************************************************************************/
   printf("Window size = %d (You can increase the window size (only odd
numbers) for more smoothing of the image) \nTime taken by entire parallel
region for median filtering = %lf \n", n, (t2-t1));

   printf("Check the local folder for the MedianFiltered_Image.pgm \n");



/**************************************************************************
   * Write the filtered image out to a file.

**************************************************************************/
  if(flag==1)
   {
      if((write_pgm_image(outputfilename, filteredimage1, rows, cols,
(unsigned char *)NULL, 255)) == 0)
        exit(1);
      free_image(image1, rows);
      free_image(filteredimage1, rows);
   }
  if(flag==2)
   {
      if((write_pgm_image(outputfilename, filteredimage2, rows, cols,
(unsigned char *)NULL, 255)) == 0)
        exit(1);
      free_image(image2, rows);
      free_image(filteredimage2, rows);
   }
 if(flag==3)
   {
      if((write_pgm_image(outputfilename, filteredimage3, rows, cols,
(unsigned char *)NULL, 255)) == 0)
        exit(1);
      free_image(image3, rows);
      free_image(filteredimage3, rows);
   }
 if(flag==4)
   {
      if((write_pgm_image(outputfilename, filteredimage4, rows, cols,
(unsigned char *)NULL, 255)) == 0)
        exit(1);
      free_image(image4, rows);
      free_image(filteredimage4, rows);
   }
 if(flag==5)
   {
      if((write_pgm_image(outputfilename, filteredimage5, rows, cols,
(unsigned char *)NULL, 255)) == 0)
```

```
        exit(1);
      free_image(image5, rows);
      free_image(filteredimage5, rows);
   }

}

/***************************************************************************
***
* Function: median_filter
* Purpose: This function will median filter an image using an n x n window.
****************************************************************************
**/
void median_filter(unsigned char **image, int rows, int cols, int n, unsigned
char ***filteredimage, int thread_count)
{
   unsigned char *pixel_values=NULL;
   int r,t, c, rr, cc, p,my_rank;
   double t3,t4;
   double *time = (double*) malloc((thread_count)*sizeof(double));
   for(t=0;t<thread_count;t++)
     time[t] = 0.0;


/***************************************************************************
   * Allocate an array to store pixel values. There will be up to n x n pixel
   * values to sort at each pixel location in the image.

***************************************************************************/
   if((pixel_values = (unsigned char *) malloc((n*n) * sizeof(unsigned
char))) == NULL){
     fprintf(stderr, "Error allocating an array in median_filter().\n");
     exit(1);
   }


/***************************************************************************
   * Allocate the filtered image.

***************************************************************************/
   if(((*filteredimage) = allocate_image(rows, cols)) == NULL) exit(1);


/***************************************************************************
   * Scan through the image and compute the median of the local pixel values
   * at each pixel position.

***************************************************************************/
```

```
    # pragma omp parallel for collapse(2) ordered num_threads(thread_count)
default(none) shared(image, filteredimage, rows,cols,n,time)
firstprivate(pixel_values) private(my_rank,t3,t4,p,rr,cc) schedule(dynamic,1)
    for(r=0;r<rows;r++)
    {
      for(c=0;c<cols;c++)
      {
         my_rank = omp_get_thread_num();
         t3 = omp_get_wtime();
         p=0;
         for(rr=(r-(n/2));rr<(r-(n/2)+n);rr++)
         {
           for(cc=(c-(n/2));cc<(c-(n/2)+n);cc++)
           {
              if((rr>=0)&&(rr<rows)&&(cc>=0)&&(cc<cols))
              {
                 pixel_values[p] = image[rr][cc];
                 p++;
              }
           }
         }

         /*******************************************************************
         * Sort the array of pixels. Although there can be up
         * to n x n pixels in the array, there are actually only p values.
         *******************************************************************/
        #pragma omp critical
        {
         insertion_sort(pixel_values, p);
        }

/*********************************************************************
         * Assign the median pixel value to the filtered image.

*********************************************************************/
         #pragma omp ordered
         {
           (*filteredimage)[r][c] = pixel_values[p/2];
         }
         t4= omp_get_wtime();
         time[my_rank]+=(t4-t3);
      }

   }
   for(t=0;t<thread_count;t++)
     printf("Time taken by thread %d to compute its chunk = %lf
\n",t,time[t]);
   free(pixel_values);
}
```

```
void insertion_sort(unsigned char *item, int count)
   {
    int c,d,t;
    for (c = 1 ; c <= count; c++)
    {
      d = c;
      while ( d > 0 && item[d-1] > item[d])
      {
        t = item[d];
        item[d] = item[d-1];
        item[d-1] = t;
        d--;
      }
    }
  }
```

2. imageio.c

```
/*****************************************************************************
***
* Program: imageio.c
* Purpose: This souce code file contains functions for dynamically allocating
* and freeing 8-bit (unsigned char) images. It also contains functions for
* reading and writing images to files in raw PGM format. This code was
written
* to be used as a teaching resource.
* Name: Michael Heath, University of South Florida
* Date: 1/7/2000
*****************************************************************************
**/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "imageio.h"

/*****************************************************************************
***
* Function: allocate_image
* Purpose: This function allocates an image. The image is an array of
pointers
* to arrays. The array of pointers will have a length of the number of rows,
* and each of these pointers will point to a separate one dimensional array
* whose length is the number of columns in the image. This scheme was used
* because it allows the image to be accessed using the syntax image[r][c]
* yet still allow the image to be any size.
* Name: Michael Heath, University of South Florida
```

```
* Date: 1/7/2000
*************************************************************************
**/
unsigned char **allocate_image(int rows, int cols)
{
   unsigned char **image=NULL;
   int r, br;


/************************************************************************
   * Allocate an array of pointers of type (unsigned char *). The array is
   * allocated to have a length of the number of rows.

************************************************************************/
   if((image = (unsigned char **) calloc(rows, sizeof(unsigned char
*)))==NULL){
      fprintf(stderr, "Error allocating the array of pointers in
allocate_image().\n");
      return((unsigned char **)NULL);
   }


/************************************************************************
   * For each row, allocate an array of type (unigned char).

************************************************************************/
   for(r=0;r<rows;r++){
      if((image[r] = (unsigned char *) calloc(cols, sizeof(unsigned
char)))==NULL){
         fprintf(stderr, "Error allocating an array in allocate_image().\n");
         for(br=0;br<r;br++) free(image[br]);
         free(image);
         return((unsigned char **)NULL);
      }
   }

   return(image);
}

/************************************************************************
***
* Function: free_image
* Purpose: This function frees the memort that was previously allocated to
* store an image.
* Name: Michael Heath, University of South Florida
* Date: 1/7/2000
*************************************************************************
**/
void free_image(unsigned char **image, int rows)
{
```

```
   int r;

/*************************************************************************
   * Free each row of the image.

*************************************************************************/
   for(r=0;r<rows;r++) free(image[r]);


/*************************************************************************
   * Free the array of pointers.

*************************************************************************/
   free(image);
}

/*************************************************************************
**
* Function: read_pgm_image
* Purpose: This function reads in an image in raw PGM format. Because the PGM
* format includes the number of columns and the number of rows in the image,
* these are read from the file. Memory to store the image is allocated in
this
* function. All comments in the header are discarded in the process of
reading
* the image. Upon failure, this function returns 0, upon sucess it returns 1.
* Name: Michael Heath, University of South Florida
* Date: 1/7/2000
*************************************************************************
*/
int read_pgm_image(char *infilename, unsigned char ***image, int *rows,
    int *cols)
{
   FILE *fp;
   int r;
   char buf[71];


/************************************************************************
   * Open the input image file for reading. If the file can not be opened for
   * reading return an error code of 0.

************************************************************************/
   if((fp = fopen(infilename, "r")) == NULL){
      fprintf(stderr, "Error reading the file %s in read_pgm_image().\n",
         infilename);
      return(0);
   }
```

```
/***************************************************************************
   * Verify that the image is in PGM format, read in the number of columns
   * and rows in the image and scan past all of the header information.

***************************************************************************/
   fgets(buf, 70, fp);
   if(strncmp(buf,"P5",2) != 0){
      fprintf(stderr, "The file %s is not in PGM format in ", infilename);
      fprintf(stderr, "read_pgm_image().\n");
      fclose(fp);
      return(0);
   }
   do{ fgets(buf, 70, fp); }while(buf[0] == '#');  /* skip all comment lines
*/
   sscanf(buf, "%d %d", cols, rows);
   do{ fgets(buf, 70, fp); }while(buf[0] == '#');  /* skip all comment lines
*/


/***************************************************************************
   * Allocate memory to store the image.

***************************************************************************/
   if(((*image) = allocate_image(*rows, *cols)) == NULL) return(0);


/***************************************************************************
   * Read in the image from the file, one row at a time.

***************************************************************************/
   for(r=0;r<(*rows);r++){
      if((*cols) != fread((*image)[r], 1, (*cols), fp)){
         fprintf(stderr, "Error reading the image data in
read_pgm_image().\n");
         fclose(fp);
         free_image((*image), *rows);
         return(0);
      }
   }

   fclose(fp);
   return(1);
/****************************************************************************
**
* Function: write_pgm_image
* Purpose: This function writes an image in raw PGM format. A comment can be
* written to the header if coment != NULL. If there is a comment, it can
* be up to 70 characters long.
* Name: Michael Heath, University of South Florida
```

```
* Date: 1/7/2000
******************************************************************************
*/
int write_pgm_image(char *outfilename, unsigned char **image, int rows,
    int cols, char *comment, int maxval)
{
   FILE *fp;
   int r;


/****************************************************************************
   * Open the output image file for writing.

****************************************************************************/
   if((fp = fopen(outfilename, "w")) == NULL){
      fprintf(stderr, "Error writing the file %s in write_pgm_image().\n",
         outfilename);
      return(0);
   }


/****************************************************************************
   * Write the header information to the PGM file.

****************************************************************************/
   fprintf(fp, "P5\n");
   if(comment != NULL)
      if(strlen(comment) <= 70) fprintf(fp, "# %s\n", comment);
   fprintf(fp, "%d %d\n", cols, rows);
   fprintf(fp, "%d\n", maxval);


/****************************************************************************
   * Write the image data to the file.

****************************************************************************/
   for(r=0;r<rows;r++){
      if(cols != fwrite(image[r], 1, cols, fp)){
         fprintf(stderr, "Error writing the image data in
write_pgm_image().\n");
         fclose(fp);
         return(0);
      }
   }

   fclose(fp);
   return(1);
}
```

3. imageio.h

```
/*****************************************************************************
***
* Program: imageio.h
* Purpose: This header file contains function prototypes for functions that
* dynamically allocate and free 8-bit (unsigned char) images. It also
contains
* prototypes for functions that read and write images to files in raw PGM
* format. This code was written to be used as a teaching resource.
* Name: Michael Heath, University of South Florida
* Date: 1/7/2000
*****************************************************************************
**/
#ifndef _PGMIO_
#define _PGMIO_

unsigned char **allocate_image(int rows, int cols);

void free_image(unsigned char **image, int rows);

int read_pgm_image(char *infilename, unsigned char ***image, int *rows, int
*cols);

int write_pgm_image(char *outfilename, unsigned char **image, int rows,
   int cols, char *comment, int maxval);

int write_gray_bmp(char *outfilename, unsigned char **image, short int rows,
short int cols);
#endif
```

4. median_serial.sh

```
#!/bin/bash

CC=gcc
EXEC=median
SRC=median_serial.c
COMP=imageio.c
IN=Input_Image1.pgm
OUT=MedianFiltered_Image_Serial.pgm

if [ "$SRC" -nt "$EXEC" ]
then
   echo "Compiling..."
```

```
   $CC -o $EXEC $SRC $COMP
fi

./$EXEC -n 9 $IN $OUT
```

5. median_parallel.sh

```
#!/bin/bash

CC=gcc
EXEC=median
SRC=median_parallel.c
COMP=imageio.c
IN1=Input_Image1.pgm
IN2=Input_Image2.pgm
IN3=Input_Image3.pgm
IN4=Input_Image4.pgm
IN5=Input_Image5.pgm
th=16

OUT=MedianFiltered_Image_Parallel.pgm

if [ "$SRC" -nt "$EXEC" ]
then
   echo "Compiling..."
   $CC -o $EXEC $SRC $COMP -fopenmp
fi

./$EXEC -n 9 $IN1 $IN2 $IN3 $IN4 $IN5 $OUT $th
```

B. Hpc-class's compute node's architecture:

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 16
On-line CPU(s) list: 0-15
Thread(s) per core: 1
Core(s) per socket: 8
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 45
Model name: Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
Stepping: 7
CPU MHz: 1217.895
CPU max MHz: 2800.0000
CPU min MHz: 1200.0000
BogoMIPS: 4000.38
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K
NUMA node0 CPU(s): 0-7
NUMA node1 CPU(s): 8-15

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 32735 MB
node 0 free: 31611 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 32768 MB
node 1 free: 31639 MB
node distances:

node	0	1
0:	10	21
1:	21	10

C. Memory Usage:

As I could not find a very lasrge data set of .pgm images, I had to download large resolution .jpg images and convert them online to .pgm image format. Due to this, I was unable to get large chunk of data set which would have increased my data-set size.

As for now, I have used less than 1 gigabyte of memory including the five input images, the five output images and some additional parameters.

REFERENCES

[1] https://www.mathworks.com/help/images/ref/medfilt2.html

[2]https://www.cs.auckland.ac.nz/courses/compsci373s1c/PatricesLectures/Image%20Filtering_2up.pdf

[3] An Improved Median Filtering Algorithm for Image Noise Reduction - Youlian Zhu, Cheng Huang

(https://ac.els-cdn.com/S1875389212005494/1-s2.0-S1875389212005494-main.pdf?_tid=5d583886-669c-4c49-a8df-cec956c8c851&acdnat=1540960253_d76b3e6405bd110c4c0f65251d38deaf)

[4] https://www.ece.rice.edu/~wakin/images/

[5] https://www.cs.cmu.edu/~eugene/teach/algs00a/progs/

[6] https://www.youtube.com/watch?v=APkdYObUJRU

[7] Data-set source: https://unsplash.com/search/photos/4k
Converted .jpg to .pgm image from: https://convertio.co/jpg-pgm/

[8] https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[9] https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf