ParallelPic

The Imagelib processing library for c++ paralellized

Erick Eduarte Rojas B22305
Luis Felipe Rincón Riveros B25530

December 13, 2013

Abstract

This document represents a tecnical report for the image processing library "imagelib" developed for c++ and it's parallelization. You will find the manipulation of programs with OpenMP and OpenMPI.

1 Introduction

This project proposal is to parallelize and already existing c++ library that performs basic functions of image filtering. This propose was achived by using two different paradigms: OpenMP and OpenMP.

Basically, parallel computing is the simultaneous use of multiple compute resources to solve a computacional problem. There are two important part in parallelism: the computacional problem and the compute resources. The first one should be able to be broken apart into discrete pieces of work that can be solved simultaneosly, execute multiple program instructions at any moment in time and be solved in less time with compute resources than with a single compute resource. And the compute resources might be a single computer with multiple processors, an arbitrary number of computers connected by network or a combination of both.

Parallel computing is a very important tool that allows us to modelate, simulate and comprehend better the real world complex phenomena. There are some fields in wich parallelization plays an important roll. In science and engineering has been considered to be "the high end of computing", and has been used to model difficult problems in many areas like Physics, Bioscience, Genetics, Chemestry, Geology and among others. On the industrial and commercial area, it is neccesary to process large amouns of data in sophistacated ways, for example handaling Databases, Web search engines, Medical imaging and diagnosis, financial and economic modeling and advanced graphics and virtual reality.

There are some important adventages that brings using parrallel computing. One of them is saving time and money by using a little more resources that at the end will save some time and money. Also, it became possible to solve larger and complex problems. Provides concurrency and enable the use of non-local resources.

2 Objectives

2.1 General

- Parallelize some ImageLib functions whit OpenMP and OpenMPI paradigms for later comparation.

2.2 Specific

1. Analize efficiency on ImageLib functions for digital image processing.
2. Implement OpenMPI and OpenMP as a tool to improve some image processing filters.
3. Investigate about mechanism to analize the parallelized functions efficiency.
4. Compare the sequential and parallel function efficiency and also between OPenMP and OpenMPI paradigms.

3 Justification

The digital image proccesing represents an important field for the development of different areas. This is because it becames possible to extract and obtain information that can be manipulated. Of course, it becames necessary to devolps functions that allows to simulate the human vision perception and also perceives hidden data that for the human is impossible to recognize with their visual sense. However, the image computing can operate on images and preform different functions.

However, some of the functions that can be implemented to process an image consume a lot of time. It is possible to solve this problem with the parallelization of the source code, which is based on using more resources of the computer to do a job, to get the maximum hardware performance.

Also the parallel programming allows us to implement high computer preformance wich can solve some problems that are uncomputable using a secuential machine. Basically there are two types of parallel programming: using shared memory like OpenMP and using distributed memory as OpenMPI. The distributed memory gave way to the computer clusters that are many computers linked (nodes) that can work together or individually. Also in a single computer or in a node of a cluster, we can use the multiple core and multiple threads of a processor (shared memory), which has a higher speed to access the memory, improving at maximum the performance.

4 Theoretical Framework

4.1 Thread

A thread is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. Usually threads are contained in a processor or with the newest architectures in the processor cores.

In threads of the same core/processor the memory is shared, so if we try to modify the same part of the memory with more than one thread at the same time, could have some problems. Also using threads there are race conditions, this means that threads compete between them to do first the task.

4.2 Core

A central processing unit (CPU), is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system. On modern architectures a processor has multiple CPUs in a single chip, those chips are called multi-core processors, that improve the performance of the computer.

4.3 Shared Memory Model

Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors. In parallel programming the threads use the shared memory model, the thrreads of the same processor core has the same memory.

4.4 Thread Based Parallelism

OpenMP programs accomplish parallelism exclusively through the use of threads. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is. Threads exist within the resources of a single process. Without the process, they cease to exist. Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

4.5 Explicit Parallelism

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. Parallelization can be as simple as taking a serial program and inserting compiler directives. Or as complex as inserting subroutines to set multiple levels of parallelism.

5 OpenMP

OpenMp consists of a set of compiler #pragmas that control how the program works. the pragmas are designed so that even if the compiler does not support them, the programs will still yield correct behavior, but without any parallelism.

5.1 Sintax

All OpenMP directives in C++ are indicated with a sentence beggining with #pragma omp followed by parameters. After this, it is possible to include a valid OpenMP directive, then a clause (optional) and last of all, the block of code enclosed by this directive.

5.1.1 Parallel pragma

The statement #pragma omp parallelblock is used to initialize a parallel block. It means this line creates a team of threads, wich can be specified by the programmmer, and executes the operations included. After this pragma, the threads join back into one.

5.1.2 "for" Directive

The for directive especifies that the iterations of the loop inmediately following it must be executed in parallel by the team.

Clauses

Schedule: Describes how iterations of the loop are divided among the threads in the team.

- Static: loop iterations are divided into pieces of size chunk and then statically assigned to threads. The default amount chunk depends on the number of threads wich divide evenly the work.
- Dynamic: Loop iterations are divided into pieces of size chunk, and dynamically assigned annother.
- Guided: Iterations are dynamically assigned in blocks as threads request them until no blocks remains to be assigned.Similar to the one befor, but the block size drcreases each time a parcel of work is given toa thread
- Runtime: the scheduling desicion is deferred until the runtime.
- Auto: The scheduling desicion is delegated to the compiler and the runtime system.

Ordered: Specifies that the iterations must be executed as they would be in a serial program.

5.1.3 Synchronization Constructs

Master Directive: The MASTER directive specifies a part of the code that can only be executed by the master thread of the team. The rest of threads skip this instruction.

Critical: This directive specifies a region of code wich can only be executed by a thread at a time. It means that if one thread arrives to this area and other thread is executing this operation, the first thread will wait util the other finish its work.

Barrier: Synchronize all threads by making a barrier on the code. When a barrier is reached by all the threds the all can continue, else, they have to wait for the others to finish.

Atomic: Specifies that a specific memory location mest be updates atomically, rather than letting multiple threas attemp to write it.

Ordered: This directive implies that threads will need to wait before executing their chunk iterations.

5.1.4 Data Scope Atributtes Clauses

For the model of parallelization by which is defined the paradigm OMP (shared memory programming), is important to understand the data scoping.

By default, most variables on the paradigm are defines like shared, but theres also the private clause which can include loop index variable and stack variable in subroutines called from parallel regions.

This clauses define which variables should be visible to all threads in the parallel sections and which ones will be privatelly allocated.

Private: Declares variables privates to each thread. It means a new object of the same type is declared once for each thread in the team. All references to the original object are replaced with references to the new object.

Shared: This clause declare variables to be shared among all threads, which can access to the address to read and write the object.

5.1.5 Routines

- omp_set_num threads: set the number of threads that will be used on the next parallel section.
- omp_get_num_thread: Returns the number of threads that are executing the parallel region from wich is called.
- omp_get_max_threads: Returns the maximum value that can be returned by the get number of threads.

6 OpenMPI

OpenMPI is based in the use of distributed memory, where each processor could have its own work direction, and use message passing, that makes the access to local memory faster than remote memory, an example of this is a cluster where all the processors works in parallel, and communicates with a local network, however, we could use a single computer with multiple cores.

6.1 Sintax

In OpenMPI all instructions begin with MPI_, also the types of variables inits with that. To use OpenMPI we need to write #include ¡mpi.h¿ at the begin of the program, and to compile in c++, intead to use g++,it use mpiCC, mpic++, or mpicxx.

Communicators: MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.The predermitated communicator is MPI_COMM_WORLD that communicates all processors of the computer.

Rank: Within a communicator, every process has its own, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a taskID. Ranks are contiguous and begin at zero.

6.1.1 Routines

MPI_Init: Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. And it use the same parameters of main function. Example: MPI_Init(&argc, &argv).

MPI_Comm_size: Returns the total of processes in a communicator, that represents the total MPI tasks available. Example: MPI_Comm_size (comm,&size).

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator.Example: MPI_Comm_rank (comm,&rank)

MPI_Get_processor_name: Returns the processor name. Also returns the length of the name. Example: MPI_Get_processor_name (&name,&resultlength).

MPI_Wtime: Returns an elapsed wall clock time in seconds (double precision) on the calling processor.Example: MPI_Wtime().

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program, no other MPI routines may be called after it. Example: MPI_Finalize().

MPI_Send: Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Example:
MPI_Send (&buf,count,datatype,dest,tag,comm)
Buf is the direction of the data to be send. Count the number of elements to send. Datatype is the MPI type of data (Example: MPI_INT). Dest is the destiny variable of the data. Tag is an identifier of the message, and comm is the communicator.

MPI_Recv: Receive a message and block until the requested data is available in the application buffer in the receiving task.
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
Count is the number of elements received. Source is the origin of the message. For a receive operation, indicates the source of the message and the tag of the message.

MPI_Barrier: Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed. Exampe: MPI_Barrier (comm).

MPI_Bcast: Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group. Example: MPI_Bcast (&buffer,count,datatype,root,comm).

MPI_Scatter: Data movement operation. Distributes distinct messages from a single source task to each task in the group. Example: MPI_Scatter (&sendbuf,sendcount,sendtype,&recvbuf, recvcount,recvtype,root,comm).

MPI_Gather: Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.
Example: MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm).

MPI_Reduce: Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task. Example: MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
Op is the MPI operation to do at the elements of message. Example: MPI_MAX

7 Performance Metrics for Parallel Algorithms

7.1 Speed up and efficiency

The speed up refers to the ratio of execution time of the serial code and runtime code in parallel.

$$S_P = \frac{T_s}{T_P}$$

1. Relative speed up: The sequential time is considered the execution time of the parallel program when it is executed on a single processor.

   $$Relatedspeedup(I, P) = \frac{T(I,1)}{T(I,P)}$$

   Where I is the instance of the data size given as parameter and P the number of used processors.

2. Real Speed up: The sequential execution time used is the time of the better program to solve the problem.

The efficiency of parallelization is refered to the ratio between speed up and the number of processors.If this parameter is closely to 1, the parallelization is ideal.

$$E_P = \frac{S_P}{P} = \frac{T_S}{PT_P}$$

It's a relative measure that allows comparison of performance on different parallel computing environments.

7.2 Amdahl's Law

It is important to emphasized that all parallel algorithm has a sequencial part wich limits the execution time reduction. The Amdahl law dictates that the computing time to do some operation with P processors is:

$$S_P = \frac{T_S}{T_P} \leq \frac{1}{f_s + \frac{f_p}{P}}$$

Where f_s is the sequential part of the code and f_p is the parallel part of the code. Generally the Amdahl's Law gives an upper bound to the maximum speed up and in the most real cases exists some factors that don't alow to be reached the ideal speed up, like in input/output data, waiting another proccess, communication costs...

7.3 Scalability

In scalables systems, the efficiency is constant independent of the amount of work and operations executed to solve a problem, and the number of processors or the hardaware. It gives a parameter that indicates how parallelized or scalable an algorithm is.

8 Functions

Here are some functions analysis of filters wich have been parallelized and mesured on thime of execution. The analysis porpuse is to compare the two paradigm and it's sequential time.

With OpenMP the parallelizaion was made by the "for" direcrive and the ordered clause. The for direcive splits the number of chunk by thread, depending in how many threads the user wants to use. This for directive was designed for the image height and the chunk by the image size per thread. So the height iterations are divided , but all threads execute the width iterations to apply the filter all over the image. Inside of the four dimentions the function to set a pixel may be executed as it was a sequential order, so the ordered clause is used. This is because , synchronization between threads becames very important at the time to write on the image, so the process won't be affected.

With OpenMPI ocurred a significant error with the communication between processors and the image class objects. So it became necessary to convert the image into a sequencial array of primitive types that the broadcast , scatter and gather will support. Then, converted into an array of integers by all, each processor recieves a part of the array depending, as designed OpenMP, on the number of processors used by the user. Once all of them have a local array, each of them applies the funcion to the picture and then gather it. The last step is to reconstruct the sequencial array into an image object.

Below you will find the comparison between some functions programmed, all of them with both paradigms and in the way described before.

8.1 Image addition

The addition of two images sums the pixel values of two images with the same dimensions, is useful to detect the differences between two images wich has been filtered.

Computing the speedup of the best parallel time with OpenMPI:

$$S_P = \frac{T_s}{T_P} = \frac{0.41}{0.41} = 1$$

Computing the efficiency:

$$E_P = \frac{S_P}{P} = \frac{1}{2} = 0.5$$

Computing the speedup of the best parallel time with OpenMP:

$$S_P = \frac{T_s}{T_P} = \frac{0.354961}{0.416546} = 0.852$$

Computing the efficiency:

$$E_P = \frac{S_P}{P} = \frac{0.852}{2} = 0.426$$

The addition implementation with OpenMP, and the other ones, do not improve the execution time and this can be explained with the amdahl's law. This law stablish that a program solving a large mathematical problem will typically consist of several parallelizable parts and several sequential parts. In this case, the programs solves a very simple addition and the setting of pixels

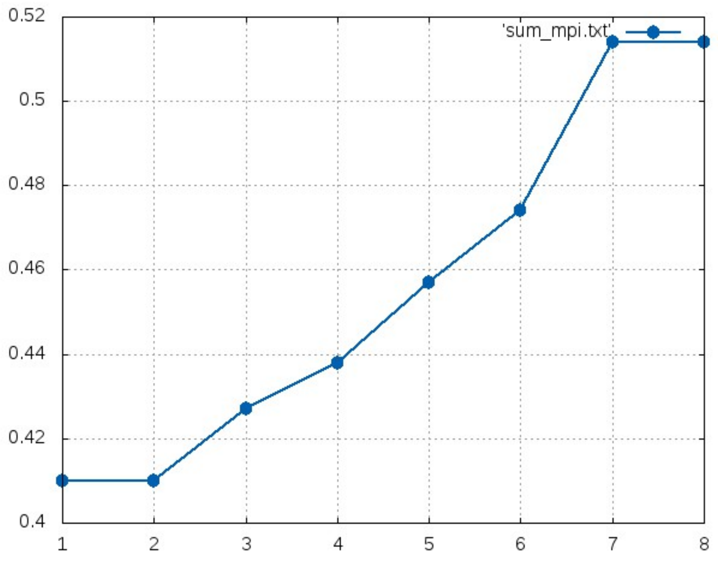

Figure 1: Addition OpenMPI time

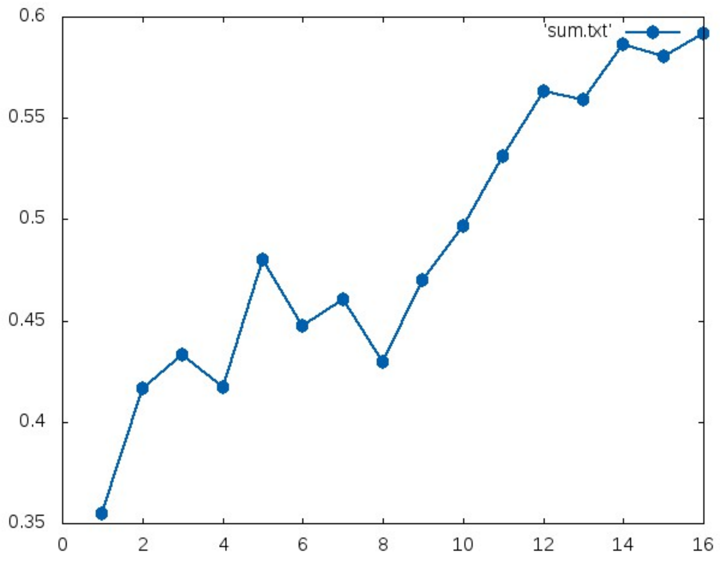

Figure 2: Addition OpenMP time

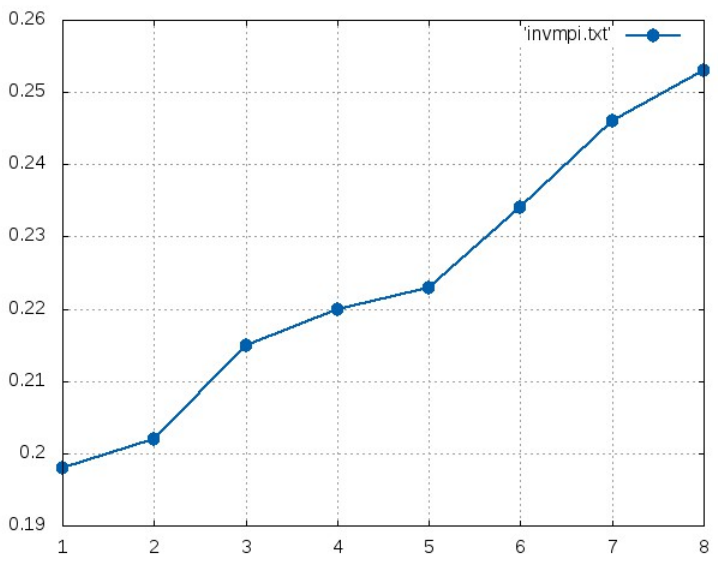

Figure 3: Inverse OpenMPI time

has to be sequential; and the parallelizable part its only the iterations all over the image. This fact limit the overall speed-up availeble from prallelizaion.

In fact not all paralization results in speed-up.Generally as a task is split up into more and more threads or processors, wich spends an everincreasing portion of their time communicating with each other. Eventually , the overhead from solving the roblem, and further parallelizaion increases rather than decreases the amount of time required to finish.

8.2 Inverse

The inverse of an image calculates the substraction of all the pixel values and the maximum value (255 to RGB), and set the new image with these pixel values.

Computing the speedup of the best parallel time with OpenMPI:

$$S_P = \frac{T_s}{T_P} = \frac{0.198}{0.202} = 0.98$$

Computing the efficiency:

$$E_P = \frac{S_P}{P} = \frac{0.98}{2} = 0.49$$

Computing the speedup of the best parallel time with OpenMP:

$$S_P = \frac{T_s}{T_P} = \frac{0.183637}{0.23025} = 0.798$$

Computing the efficiency:

$$E_P = \frac{S_P}{P} = \frac{0.798}{2} = 0.399$$

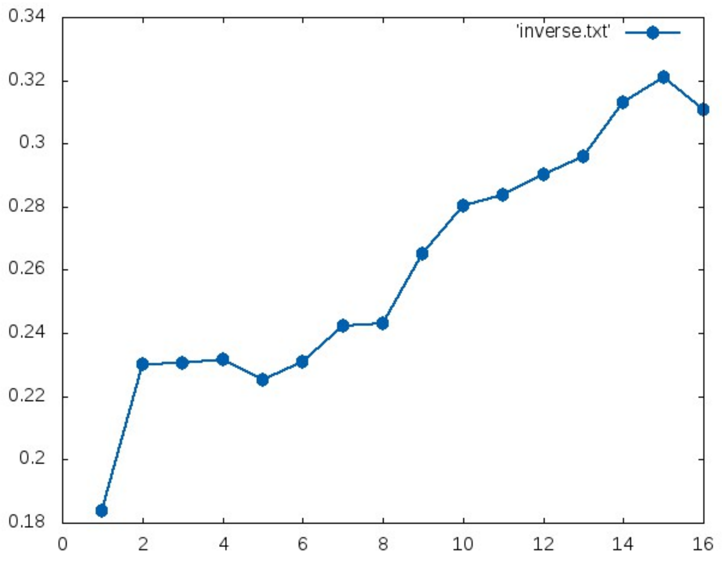

Figure 4: Inverse OpenMP time

The inverse function result with very similar times of execution between the two paradigms. The average time, taking into account both, is about 0,25s. It is possible to conclude by analyzing figure 3, that the time difference between the maximum and the minimum its only about 0.05s varying from 1 to 8 processors.

On the other hand, the implementation of OpenMP (figure 4) ilustrate how the efficiency decreases by adding more than necessary computational resources. As it is possibe to see, for more than 8 processors the execution time increases much faster than the ones befores, wich in averege executes the inverse function on 0,25s.

8.3 Horizontal Borders Filter

It's a sharpening spatial filter wich uses derivatives as their mathematical support. These kind of filters can be seen as derivatives in the x direction (for the horizontal edges enhancement).

Computing the speedup of the best parallel time with OpenMPI:

$$S_P = \frac{T_s}{T_P} = \frac{0.25}{0.25} = 1$$

Computing the efficiency:

$$E_P = \frac{S_P}{P} = \frac{1}{2} = 0.5$$

Computing the speedup of the best parallel time with OpenMP:

$$S_P = \frac{T_s}{T_P} = \frac{0.329871}{0.425047} = 0.776$$

Computing the efficiency:

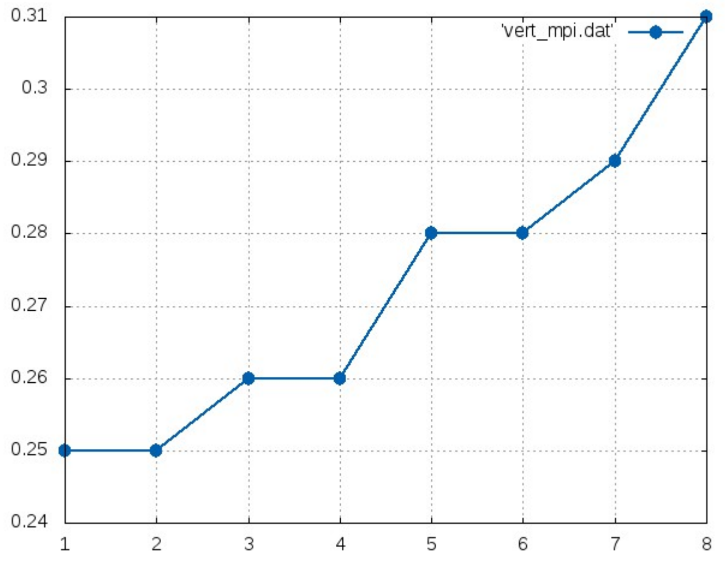

Figure 5: Horizontal Borders OpenMPI time

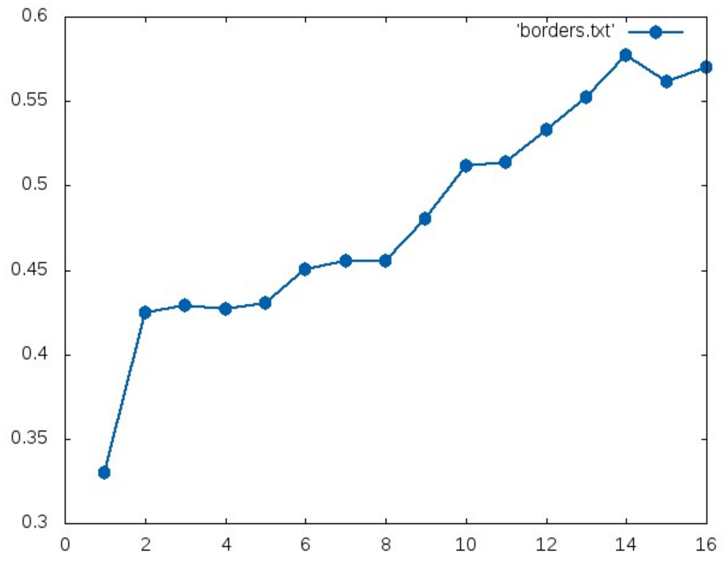

Figure 6: Horizontal Borders OpenMP time

$$E_P = \frac{S_P}{P} = \frac{0.776}{2} = 0.388$$

By looking the graphics(figure 5 and figure 6), it is possible to infer that none of parallelization can beat it's corresponding sequential time. This may be for the fact that the image processing with only one thread do not waste time on synchronization,scattering and/or gathering the image. Therefore it is hard to beat this time becuase the problem it's not so much complex.

By the other side , comparing both , OpenMPI and OpenMP, is evident that the parallelization with OpenMPI is so much efficient than the OpenMP one. Note that the more efficient time of OMP (0.33) its highest than any time measured for distributed memory programming. Nevertheless, despite the disribution of the image is made by converting it into a squencial array of primitive types and then reconstruct it to an Image object, its efficiency might be because the picture partition and merging takes lower time than threads synchronization.