Sindh Univ. Res. Jour. (Sci. Ser.) Vol.44 (3) 385-390 (2012)

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

Parallel Image Processing With Multi-Core Systems

F.A. JOKHIO, S. VIGHIO*, I. A. KOREJO**

Department of Computer Systems , Engineering, Quaid-e-Awam University, Nawabshah, Pakistan
Email: fajokhio@quest.edu.pk Email: saleem.vighio@quest.edu.pk

Received 23rd May 2012 and Revised 14th July 2012

Abstract: This paper presents the parallelization of the libpuzzle library that is used to find similar (graphical) images even after modifications. Existing sequential code in C++ is modified using Cilk Plus tool in order to execute it on multiple processing units with shared memory. As the parallel version could be run on multiple processors, the execution time is comparatively less. The results in sequential version are retained in parallel version with no change in the functionality of the application.

Keywords: Parallel programming, Multicore programming, Security system

I. INTRODUCTION

As the number of multi-processing systems is increasing, the need for software development and parallel programming is also growing. Now a day, multi-processing systems are used everywhere, for example in embedded systems, desktops, laptops and workstations (Kirk *et al.*, 2010). In past decades, single processor systems were commonly used for the processing of tasks. However, as the tasks became more complex, the need for faster multi-processing units has also increased. Existing sequential programs do not execute faster automatically on multiple processing units until the programs are simplified (or modified) (Mattson *et al.* 2004). In this paper, we present analysis of picture matching software that can take any graphical image (JPEG, PNG, and GIF) as an input and finds the similar images even if the images are later modified. The original matching program is sequential and is available as open source (Frank 2003). We modify this software in order to execute it on multi-core system using parallel programming tool Cilk++ (Intel Co. 2011a). Under its basic functionality, the program takes one source image as an input and compares it with a large set of images. The large set of images must be stored in local drive or in a network drive. It identifies the similarity between pictures. Since the machine is assumed to have more than a single core, the Cilk-based parallel version of the application is intended to exploit parallelism and focusing on scalability.

The rest of the paper is organized as follows: Section 2 gives the necessary background of the image matching technique. A detailed discussion about the Cilk++ tool is provided in Section 3. Approaches to the parallelization of picture matching application are presented in Section 4. Section 5 discusses the proposed approach. A discussion on the experimental results is presented in Section 6, and finally Section 7 gives the conclusion of the work.

2. MATERIAL AND METHODS

Background

Comparison of any random data is quite straightforward; however, it is not the case with images. Even though, a naked eye can witness the similarity between any two pictures, technically, it is hard to realize the similarity. This suggests that, there is a need to devise a method that can compare two pictures. The image signature method has been chosen from an algorithm proposed in (Wong *et al.*, 2002). Under this proposal, each image is assumed to be a matrix of pixels, stacked in order of the size NxN. With that size, the image now has N row-pixels and N column-pixels starting from 1, 2, 3... N. The image is virtually divided into smaller blocks. The normalized grey value for each such smaller block is calculated. The final signature value of the image is obtained by simply concatenating all the normalized grey values of individual smaller blocks. Now, when it comes down to comparison between two images, these signatures are the keys. Comparison between signatures of any two images is literally comparing the images with each other. Since the signatures are vectors, their comparison is represented as a distance value. The closeness of the distance value simply represents the similarity between the pictures. Both pictures might have very different resolutions, however, if there is similarity between them then their vectors will be similar and the difference between vectors will be very low.

++ Corresponding author: I. A. KOREJO Email: cskorejo@hotmail.com cell.No. +92-3323605603
*Department of Information Technology, Quaid-e-Awam University, Nawabshah, Pakistan
**Institute of Mathematics and Computer Science, University of Sindh, Jamshoro, Pakistan

2. THE TOOL CILK++

The Cilk++ (Intel Co. 2009b) (Leiserson *et al*. 1996) is a parallel programming tool released by the Intel in September 2010. Initially, it was licenced software but now a day it is available as an open source tool as well. The open source version of the tool can run on Linux. It is claimed by the manufacturer that the tool has very low overhead and performs very well and thus is suitable for parallel programming (Blumofe, 1995a). Any existing C/C++ code can be compiled and executed by the tool without any modifications. Furthermore, the tool provides also its own keywords in order to parallelize a program. The amount of speedup obtained in a parallelized implementation of an application depends on available cores, their usage and the nature of the code itself (Blumofe, 1995b). In order to realize an improved speedup, Cilk Plus tool should be tried on a machine with at least two working cores (Intel Co. 2009b).

A. Cilk++ keywords

Cilk_spawn keyword is used to indicate the beginning of parallelism. After executing this keyword, it is possible to execute the following statements in parallel along with the function which was spawned. The cilk_spawn keyword is always used before calling a function. It can have the following form:

```
/ / func ( ) returns a value
var = cilk_spawn func (args);
/ / f unc ( ) returns void
cilk_spawn func (args);
```

Here current strand may run in parallel with the function "func". The function "func" is called two times in the above section of code, first time it returns a value, whereas second time it does not. The returned value will be stored in variable "var". The type of variable "var" and the return type of function "func" should be the same. The variable "var" is known as a receiver. There is no need of any receiver for void functions. The function "func" also accepts an argument "args". There is no limit on the number of arguments. The rules for arguments passing are similar to C/C++. However, care must be taken while passing an argument by reference. The life cycle of such arguments continues until the cilk_sync keyword is executed (cilk_sync is defined below). There is a concept of parent and child functions in cilk++. If a function is spawned, it is called a child and other function in which the cilk_spawn statement is executed is known as the parent. The cilk_spawn keyword allows both child and parent functions to be executed simultaneously. The cilk++ scheduler determines dynamically what actually executes in parallel when multiple processors are available. **(Fig.1)** indicates spawn knot.

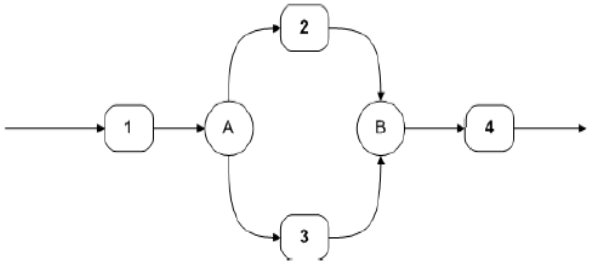


Fig.1. Parallel implementation of Cilk++

To describe a serial program, a call graph can be used, however in order to analyse a parallel program one more additional layer is added. In order to analyse the performance of a parallel program we distinguish only between serial and parallel sections of code. The serial section of code is termed as strand. The point at which two or more strands meet is termed as knot and there are two types of knots, - a spawn knot and a sync knot. Fig. 1 illustrate 4 strands (1, 2, 3, 4), a spawn knot (A) and a sync knot (B). Strands (2) and (3) may execute in parallel. This kind of structure is shown in the following code:…

```
/ / executes tran d1
do stuff1 ( ) ;
/ / spawn strand3 at knot A
cilk_spawn func3 ( );
/ / executes tran d2
do stuff2 ( ) ;
/ / sync at knot B
cilk_sync;
/ / executes trand 4
do stuff4 ( );. . .
```

Cilk_sync keyword is used to specify serialization of statements. Every parallel implementation has to be synchronized at some stage in order to terminate the application properly. Once the desired parallel implementation is achieved, cilk_sync can be specified such that the compiler is informed of the statements following cilk_sync in order to execute them serially (Intel Co. 2009b).

Cilk_for keyword is used to parallelize a task even inside a loop. Nested loops may involve larger computation which in turn leads to a larger overhead, if too many spawns are used. Instead to parallelize statements inside a loop, cilk_for can be a potential tool which can implement a number of iterations that can run in parallel. Furthermore, cilk _or has an implicit "sync" function that resolves the ambiguity on where to exactly synchronize. The cilk_for keyword employs divide and conquer strategy in order to execute iterations in parallel. Initially, the work on total number of iterations is divided to equal parts and then this work is assigned to new child workers. Each parent thus has

two child workers and each child from a parent has again two new child workers. This is represented in Fig. 2 below.

(Fig. 2) shows the division of work between child workers when a cilk_for loop is encountered. The parallel implementation of child workers is synchronized at the end of the loop. This approach makes the work to be evenly balanced so that all working cores would be used efficiently. Spawned child workers of an iteration are synced at the end of that iteration itself; this ensures that the child workers from other iterations or parallel statements outside the for loop would not be synced with them.

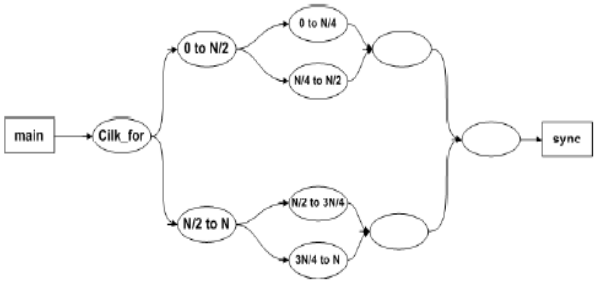


Fig.2. Parallel implementation of iterations inside a Cilk_for loop

B. Approaches To Parallelizing Picture Matching Applications

The image comparison application is assumed to have a source image and a set of target images which are to be compared with the source image. The comparison is based on the signature parameter which is to be calculated individually for each image. This indicates that the application should handle a source image as well as the set of target images. To parallelize such an application, we have the following approaches for mapping images to processors.

C. One-to-one mapping

This is a straight forward method in which each image on the target set is mapped to one of the available cores and then the signature is calculated. Under this approach, it is required that the number of target images and the number of available cores should be the same **(Fig.3).**

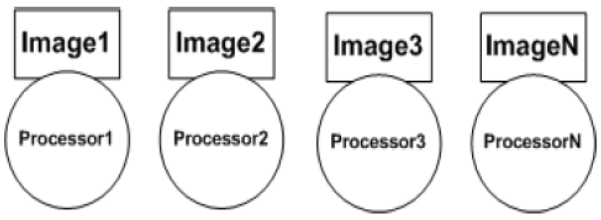


Fig.3. One-to-one mapping of images

(Fig. 3) shows the mapping of an image on a single processor. In practice, one-to-one mapping could be inefficient if the size of each particular target image is non-uniform. In fact, this leads to a state where the cores having smaller images execute the task in less time and then they are idle, whereas other processors with larger images take longer to execute the task.

D. One-to-many mapping

In this approach, the task of calculating the signature of one image is shared among a different number of processors. Assuming M number of images, N number of processors, and each task inside M^{th} image as a, b; we get each of the processors carrying one of these tasks **(Fig. 4)**.

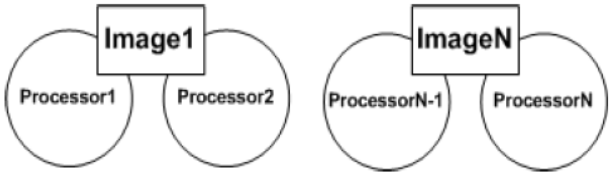


Fig.4. One-to- many mapping of images

(Fig. 4) shows an example of mapping one image to many processors. In the figure, each image is assumed to have two tasks which can actually vary depending on the type of the application. When an image has internally many tasks to perform, one image can be mapped to a number of processors which is equal to the number of tasks to be performed. Arguably, this approach is better in terms of execution time. However, in order to parallelize at image level, it may require more time and effort.

E. Many-to-many mapping

This technique allows mapping of many images to many processors. Under this approach, each processor can handle more than one image at a time. Effectively, the signatures of images mapped to different processors are computed in parallel.

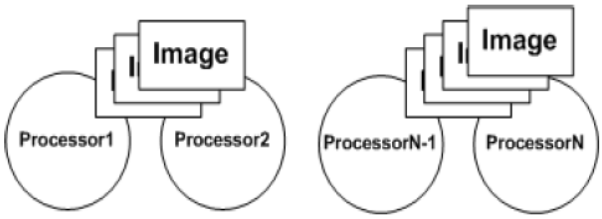


Fig.5. Many-to- Many mapping of images

(Fig. 5) shows the mapping of many images on many processors with the number of images is greater than the number processors. This approach could be more efficient to handle bulk number of target images.

F. Proposed Approach

Based on the above discussion of mapping approaches, we preferred one-to-one and many-to-many approaches for use. The selection of these approaches is ideal when the set of target images is high. The idea is to parallelize the calculation of signatures of each of the target images in parallel as they are mutually exclusive tasks. Once these signatures are available, the corresponding distance vectors are calculated. Comparison of these signatures with the reference picture gives a distance vector which defines a level of similarity between any two images.

(Fig. 6) represents the parallelized solution which calculates the signatures of each image in parallel. The normalized distances are however calculated in series with their respective signatures.

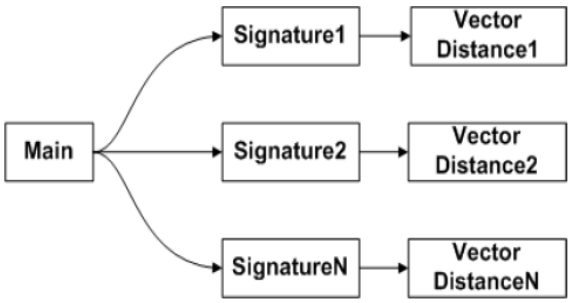


Fig.6. Proposed parallelization approach for libpuzzle

The main idea of the parallelization is to utilize available cores for processing the similarity of an image with the reference picture using cilk_for and cilk_spawn keys. The computation of signatures (puzzle vector) and normalized distances of different images can be performed in parallel, however due to interdependence among the tasks, it is not possible to calculate signature (puzzle vector) and normalized distance of the same image in parallel. For the same image, signature must be calculated and then normalized distance can be calculated. Vector distance of an image is obtained only after the computation of its signature. Literally, this means that the calculations of the signature and normalized distance should be serial. The final distance vectors are sorted in an ascending order using the basic quick sort algorithm.

The implementation of the proposed approach is possible with both cilk_spwn and cilk_for keywords. We implemented one-to-one approach with cilk_spawn. The following code indicates the implementation using cilk_spawn.

```
while ( fgets ( line, 80 , fr ) != NULL)
{
int moi = 0 ;
  for ( int moii= 0 ; moii<= ( 6 0 ) ; moii ++)
      str[moii ] = 'n0 ' ;
      temp = strlen (line) ;
      strcpy (str , " Images " ) ;
   for ( int moi= 0 ;moi<= ( temp - 39) ; moi++)
      str[moi ] = (line [37+moi ] ) ;
      opts.file2 = str ;
      strcpy (Images [index] , str ) ;
      cilk_spawn puzzle_fill_cvec_from_file
(&context,
      &cvec2 [index], opts.file2);
      index = index + 1 ;
      if( index>=numofproc ) {
      cilk_sync ;
    for(int dis = 0; dis<numofproc ; dis++)
        { d = puzzle_vector_normalized_distance
          (&context, &cvec1, &cvec2[dis],
                opts.fix_for_texts) ;
         printf ("%s %g \n" , Images [dis] , d ) ;
         { printf (fp , " %s%g \n " , Images [dis], d) ; }
          Index = 0;}
}
if (index >=0) {
     cilk_sync;
     for (int dis = 0; dis<index; dis++)
       {d = puzzle_vector_normalized_distance
          (&context, &cvec1 , &cvec2 [dis] ,
               opts.fix_fortexts) ;
        printf ("%s %g \n " , Images [dis] , d) ;
        { printf(fp , " %s %g \n " , Images [dis] , d ); }
        index = 0; }
```

The above code shows the vector calculation for the image set only. The source image vector is already calculated and the time required calculating vector for one image may differ from the other. The idea is this: each time we take the same number of images as the number of available processing cores, then we calculate the vectors in parallel until all vectors are calculated, finally we calculate the distance between those vectors. We continue in the same way until no more images are left.

The many-to-many approach uses cilk_for keyword for parallelization. Here we create large number of threads. In each thread we first calculate the puzzle vector of an image and then calculate the distance of that vector with the source image. The cilk_for implementation is shown in the following code that shows the calculation of the vector for the source image.

```
int i ;
cilk_for (i = 0; i<nfiles; i++)
   {  char full_file_name[MAX_PATH_LEN];
```

```
    sprintf(full_file_name, "%s%s ", opts.input_dir,
                    items[i ] . fname ) ;
    PuzzleCvec cvec2;
    double d;
    puzzle init cvec(&context, &cvec2 );
 int tmp = puzzle_fill_cvec_from_file (&context,
&cvec2,
                full_file_ name);
   if (tmp != 0) {
       printf (stderr, "Unable_to_read[%s]\n",

full_file_name); }
   else { d = puzzle_vector_normalized_distance
        (&context, &cvec1, &cvec2,
opts.fix_for_text);
       items[i].distance = d ;
       strcpy (items [i].fname, items[i].fname); }
       puzzle_free_cvec(&context, &cvec2 ); }
```

3. RESULTS

The experiments are performed using quad-core processor with hyper threading support. The machine has Intel (R) core(TM) i7-2630QM CPU @ 2.00 GHz processor with 6.00 GB of installed memory (RAM). We perform experiments for different image sets in order to see the functionality and speed-up of the parallel version of the code. The functionality observed is exactly the same as for the sequential version. The speed-up obtained for up to eight processor cores for different sets of image is also similar.

G. Profiling results

In order to find out the hot spots, we profile the parallel version of the application. The purpose of profiling is to find out the functions which consume most of the time in the application. We performed the profiling operation with three different sizes of images and found that the following functions are taking most of the time.

- gdImageGetTrueColorPixel
- puzzle fill dvec from file
- puzzle autocrop view
- jpeg idct 16x16

H. Speedup

(Fig. 7) shows the speed-up obtained using both approaches. The speed-up obtained using many-to-many is clearly better than one-to-one approach. This is because of the fact that in one-to-one approach, all processors get one image in order to calculate the vector, after that they wait until all vectors are calculated, then the distance between vectors is calculated and new images are assigned to them. In many-to-many approach this is not the case. Every processor can get a new image and does not wait for the other processors to complete their task. In this way, there is no waiting time for processors.

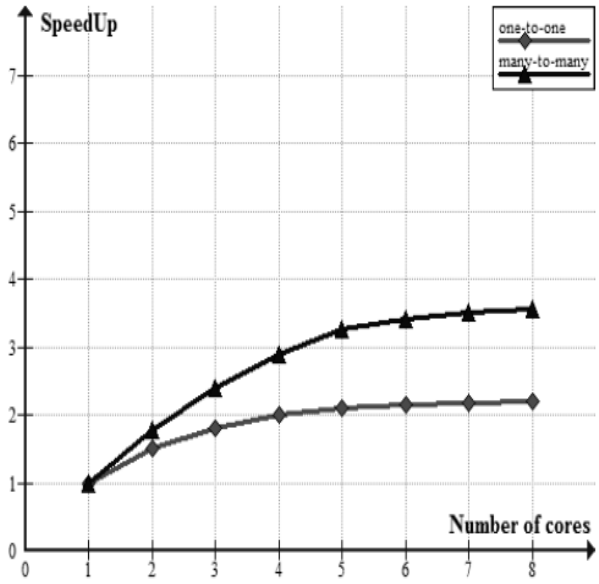


Fig.7. Speedup

4. CONCLUSION

In this paper, we provided a detailed discussion about the parallelization of libpuzzle library using the Cilk++ tool. We analysed two different approaches for parallelization: one-to-one and many-to-many. Our experimental results suggested that, the speed-up obtained using many-to-many is better than one-to-one approach. This is because of the fact that under many-to-many approach a processor does not wait for other processors to complete their work and calculates the vector of newly received (input) images without any delays. Furthermore, our analysis results also showed that the libpuzzle library is designed in such a way that the source picture of any size could be compared to any number of target pictures of any size. In addition, the code is developed in such a way that complex file names (strings) could also be read.

REFERENCES:

Blumofe R. D. (1995a) Executing multithreaded programs efficiently. PhD thesis.

Blumofe R. D., C. F. Joerg B. C. Kuszmaul C. E. Leiserson K. H. Randall and Y. Zhou (1995b) Cilk: An efficient multithreaded runtime system, in *Journal of parallel and distributed computing*, 207–216.

Frank D. (2003) The puzzle library. Online available at: http://libpuzzle.pureftpd.org/project/libpuzzle/

Intel Co. (2009b) Intel Cilk++SDK programmer's guide, document number: 322581-001US.

Intel Co. (2011a) A quick, easy and reliable way to improve threaded performance: Intel Cilk plus. Online available at: http://software.intel.com/en-us/articles/intel-cilk-plus/

Kirk D.B. and W.W. Hwu, (2010) Programming Massively Parallel Processors: A Hands-on Approach, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Leiserson C. E., R. Morgenthaler and C. F. Joerg, (1996) The cilk system for parallel multithreaded computing, *Tech. Rep*.

Mattson T., B. Sanders and B. Massingill (2004) Patterns for parallel programming, 1st ed. Addison-Wesley Professional.

Wong C., M.W. Bern and D. Goldberg (2002) An image signature for any kind of image, in *ICIP (1)*, 409–412.