Journal of Parallel and Distributed Computing 137 (2020) 134–147

Parallel Implementation of the Image Block Representation using OpenMP

Iraklis M. Spiliotis, Michael P. Bekakos, Yiannis S. Boutalis
Department of Electrical and Computer Engineering
Democritus University of Thrace
GR-67100, Xanthi, Greece

e-mails: spiliot, mbekakos, ybout{@ee.duth.gr}

Abstract

Herein, a parallel implementation in OpenMP of the Image Block Representation (IBR) for binary images, is investigated. The IBR is a region-based image representation scheme that represents the binary image as a set of non-overlapping rectangular areas with object level, called *blocks*. The IBR permits the execution of operations on image areas instead of image points and therefore leads to a substantial reduction of the required computational complexity. The experimental and the analytically derived results from parallel implementation in OpenMP, on a multicore computer, proved that a very good overall performance can be achieved.

Keywords

Image Block Representation, Image Processing, Karp-Flatt metric, Parallel Computing, Parallel Algorithms, OpenMP

1 Introduction

In our days vast amounts of data are generated, processed, analyzed and transferred. According to Cisco Inc., global IP traffic will reach 396 exabytes (EB) per month by 2022, and IP video traffic will be 82 percent of all IP traffic by 2022 [37]. Images include large amount of information, thus, the majority of image processing and

analysis techniques are not fast; the acceleration of these techniques is of great importance and vital for a number of applications.

The existence of various parallel computing platforms permits the parallel implementation of various algorithms and operations. A popular parallel platform is the Shared Memory Parallel Machine (SMPM) which is a multicore, shared memory computer which usually uses the OpenMP (Open Multi Processing) API [14]. The OpenMP parallelism is accomplished using threads, where a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler and typically each CPU core executes a thread. The OpenMP is also suitable for Intel ® Xeon Phi™ coprocessor accelerators.

A second parallel computing platform is the Distributed Memory Parallel Machine (DMPM) model, usually consisted of a cluster of standalone machines, a communication network infrastructure and the MPI API. The newer parallel computing platform is the General-Purpose Graphics Processing Units (GPGPUs) and the OpenCL API from ATI or the CUDA API created by Nvidia. All the above platforms are in service today in supercomputers and High Performance Computing (HPC) centers.

SMPM and the OpenMP API have been used for the parallel implementation of various image processing and analysis algorithms. Specifically SMPM and OpenMP have been used for the implementation of a fast neural network face detection system [15], for the parallel object identification on multi-spectral imaging [16], for the parallel implementation of the topological watershed operation for image segmentation [17], for the parallel implementation of the separation of moving objects from static background on real-time video processing [18] and for the efficient parallel implementation of Connected Component Labeling on images [19] and for the parallel Legendre moments computation [20]. Moreover, SMPM have been used for the parallel computation of the local image features SIFT and SURF using p-thread library [21].

In this paper, an image representation scheme which is called Image Block Representation (IBR) and its parallel algorithm is investigated, which is suitable for implementation on SMPMs with the OpenMP API. The proposed parallel implementation of IBR, provides a basis for the development of parallel issues of other image processing and analysis algorithms for block represented images.

2 Related Work

The classical image representation format is the 2-D array, where the value of an array element represents the intensity of the respecting pixel. However, many research efforts to derive alternative image representations have been motivated by the need for fast processing of huge amounts of data. Such image representation approaches aim to provide machine perception of images in pieces larger than a pixel and are separated into two categories: *region*-based methods and *boundary*-based methods. The *region*-based representations include quadtree [1], run length encoding [2], [3] and interval coding representation [4]. The *boundary*-based representations include chain code [5], contour control point models [6] and autoregressive models [7]. In the context of binary images, the run length and interval coding representations are identical. A region-based method, called IBR has been studied in the past [8]. In the IBR process the whole binary image is decomposed into a set of rectangular areas with object level, called *blocks* and the fact that many compact areas of a given binary image have the same value is exploited.

As in every other region-based method, the most important characteristic of the IBR is that a perception of image parts greater than a pixel is provided to the machine and therefore, all the operations on the pixels, belonging to a block, may be substituted by a simple operation on the block. This is a key point for block represented images and the IBR process may be considered as the creation of an intrinsic data parallelism, in comparison with the classical 2-D array image representation.

Taking this feature into account, the implementation of new algorithms for binary image processing and analysis operations, leads to a substantial reduction of the required time and computational complexity. Based on this representation, the fast shift, scale and rotation, connected component labeling, logic operations [8], the real time moments computation [9], a fast parallel skeletonization algorithm [10], a fast thinning algorithm [11] and a fast algorithm for the computation of the Hough transform [12] have been studied. The IBR scheme has been also used for gray image analysis and specifically for the real time computation of image moments on a serial computer [13].

3 Image Block Representation

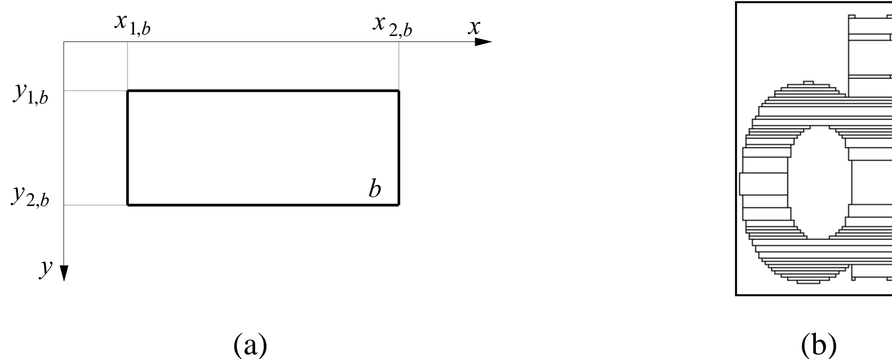
Suppose that in a binary image the object pixels are assigned to level 1 and the background pixels are assigned to level 0 and that the object pixels are represented by a set of non-overlapping rectangles with edges parallel to the axes which are called blocks. It is always feasible to represent a binary image with a set of blocks with object level; this representation is called Image Block Representation (IBR)[8].

A binary image is called *block represented,* if it is represented by a set of blocks with object level, and if each pixel of the image with object value belongs to one and only one block. The IBR is an information lossless representation.

A block represented image is denoted as the set of the blocks, where each block is described by four integers, the coordinates of the upper left and down right corner in vertical and horizontal axes as shown in Fig. 1 (a). A block represented image is denoted as:

$$\begin{aligned} f(x,y) &= \{b_i : i = 0,1,\ldots,k-1\} \\ b_i &= (x_{1,b_i}, x_{2,b_i}, y_{1,b_i}, y_{2,b_i}) \end{aligned} \tag{1}$$

where k is the number of the blocks.



(a) (b)

Fig. 1. (a) A block *b*. (b) Image of the character d and the blocks.

Considering a WxL binary image $f(x,y)$, x=0,1, ... ,W-1, y=0,1, ... ,L-1, the block extraction process requires a pass from each line y of the image. In this pass all object level intervals are extracted and compared with the previous extracted blocks. In the following, an IBR algorithm is given.

Algorithm 1. *Serial Image Block Representation*

```
1    kp:= 0; blockno:=0;
2    for (y:=0; y<L; y++){
3      kc:=0; intervalfound:=0; j_last:=0; j_curr:=0;
4      for (x:=0; x<W; x++){
5        try2match:=0;
6        if (img[y][x] && !intervalfound)
```

```
7              {intervalfound:=1;   x1:=x; }
8            if (!img[y][x] && intervalfound)
9              { intervalfound:=0;  x2:=x-1;
10             try2match:=1; }
11           if ( x==W-1 &&  img[y][x]  &&  intervalfound)
12                   { x2:=x;  try2match:=1; }
13           if (try2match) {
14             intervalmatched:=0;
15             for (j:=j_last; j<kp   &&   x1>=block[p[j]].x1 ;  j++)
16               if (x1==block[p[j]].x1  &&  x2==block[p[j]].x2) {
17                 c[kc]:=p[j];   block[p[j]].y2:=y;
18                 intervalmatched:=1;  j_curr := j;
19               } // end if
20             j_last := j_curr;
21             if (! intervalmatched){
22               block[blockno].x1:=x1; block[blockno].x2:=x2;
23               block[blockno].y1:=y; block[blockno].y2:=y;
24               c[kc]:=blockno++;
25             } enf if (!intervalmatched)
26             kc++;
27           }// end if try2match
28         } end for x loop
29         for (i:=0;i<kc;i++) p[i]=c[i];
30         kp:=kc;
31       }// end for y loop
```

In the Algorithm 1, the input is the array *img*[][] which contains the input image and the variables L, W are the length and width of the image in pixels. The output of the Algorithm 1 is the vector *block*[] and the integer variable *blockno* which indicates the number of the extracted blocks. In Fig. 1(b) the blocks that represent an image of character d, as extracted when using Algorithm 1, are illustrated.

Usually the number of blocks in a binary image of size WxL containing one or two objects, is smaller than the width W or the length L of the image, as it has been shown experimentally in [8].

4 Parallel implementation of IBR

For the creation of the Parallel IBR Algorithm (PIBR), the Algorithm 1 is examined for data or task decomposition. At first a data partitioning scheme was considered, where the L image rows are partitioned among the p threads. This approach results to the following important data dependencies:

D1. A dependency appears in the boundary between two successive threads, specifically to match the intervals of first row of the bundle of rows that is

assigned to thread p_i, to the intervals of the last row of the bundle of the thread p_{i-1}.

D2. A data race condition appears with the access on the variable *blockno*, which stores the number of the extracted blocks. The number of blocks created from an image row, is the number of intervals of the image row that do not match to the intervals of the previous image row. Therefore, it is not possible to predict or to precalculate this number for each row and each thread and this results to a data race condition, since all threads should have a simultaneous access for read and write operations in order to increase the value of *blockno* for each new block.

D3. Another important issue is the arrangement of the indices of the extracted blocks which is also related with the variable *blockno*. In certain applications on block represented images, it is desirable that neighboring blocks or blocks from successive image rows to allocate successive positions in the vector *block*[]. Such applications include the connected component labeling [8], skeletonization [10] and thinning [11]. Therefore it is not allowed to shuffle the storage locations of the extracted blocks; for example, the *i*-th block starts in row 100: $block[i].y_1$=100 and the (i+1)-th block starts in row 50: $block[i+1].y_1$=50.

Due to the above dependencies the parallelization using data partitioning or concurrent task partitioning is not possible in the current form of the Algorithm 1. In order to parallelize the Algorithm 1, a divide and conquer strategy is followed: the algorithm is decomposed into more elementary parts and the parallelization of these parts is examined.

4.1 Decomposition of the serial IBR algorithm

In Algorithm 1 two discrete parts can be easily distinguished:

- **Part1**, *Interval extraction*, which extracts the object level intervals in every row of the image.
- **Part2**, *Interval matching*, which matches the extracted intervals of each row with the blocks of the previous row of the image. If an interval does not match with a block from previous row, then a new block is created.

For the implementation of the above Parts 1 and 2, it is required to define a suitable data structure for storing the output of Part 1, consisted of the object intervals, that is used as input to the Part 2. This data structure called *intervalrowt*, and contains the integer variable *irno* that stores the number of object level intervals in an image row and an array of pointers to the triples ($x1$, $x2$, b) of the coordinates of each interval on x-axis and the block in which the interval belongs. Also, an array called *ir*[] with length L is defined; each element of the array is of type *intervalrowt* and corresponds to a row of the image. This data structure is not required in Algorithm 1, since each object level interval is matched immediately after its extraction and there is no reason to store the intervals.

The decomposition of the serial Algorithm 1 leads to the following serial Algorithms, 2, 3, that implement sequentially the above Parts 1 and 2. Algorithm 2 scans each row of the input image and extracts all object level intervals.

Algorithm 2. *Serial Interval Extraction*

```
1    for (y:=0;y<L; y++){
2      intervalfound:=0;  ir[y].irno:=0;
3      for (x:=0;  x<W-1;  x++){
4              if (img[y][x]  &&  !intervalfound){
5                intervalfound:=1;
6                ir[y].x1[ir[y].irno]:=x;
7          }
8              else if (!img[y][x]  &&  intervalfound){
9                intervalfound:=0;
10               ir[y].x2[ir[y].irno++]:=x-1;
11             }
12     } // end x loop
13     if (!img[y][W-1]) {  // last column
14       if (intervalfound) ir[y].x2[ir[y].irno++]:=W-2;
15     }
16     else { // case img[y][imagewidth-1]  == 1
17       if (!intervalfound) ir[y].x1[ir[y].irno]:=W-1;
18       ir[y].x2[ir[y].irno++]:=W-1;
19     }
20   }// end y loop
```

The interval matching among consecutive image rows, results to the extraction of the blocks, as shown in Algorithm 3. To achieve this goal, a scan on the array *ir*[] is required. The intervals that have been extracted from the image row y are matched to the blocks of row y-1, if an interval from the current row is matched to a block of the

previous row, then the end of the block is in the row *y*, otherwise a new block is created.

Algorithm 3. *Serial Interval Matching*

```
1    blockno:=0;
2    for (i:=0; i<ir[0].irno; i++)    {
3      block[blockno].x1:=ir[0].x1[i];  block[blockno].x2:=ir[0].x2[i];
4      block[blockno].y1:=0;  block[blockno].y2:=0;
5      ir[0].b[i]:=blockno++;
6    } // end for i loop
7    for (y:=1;y<L; y++) {
8      j_prev:=0;  j_curr:=0;
9      for (i:=0; i<ir[y].irno; i++) { // i for current row, j for previous row
10       intervalmatched:=0;
11       for (j:=j_prev; j<ir[y-1].irno  &&  ir[y].x1[i]  >=  ir[y-1].x1[j];  j++)
12         if (ir[y].x1[i]==ir[y-1].x1[j]  &&  ir[y].x2[i]==ir[y-1].x2[j])
13           {ir[y].b[i]:=ir[y-1].b[j];  block[ir[y].b[i]].y2:=y;
14           intervalmatched:=1;  j_curr:=j; }
15       j_prev := j_curr;
16       if (! intervalmatched) {
17         block[blockno].x1:=ir[y].x1[i];  block[blockno].x2:=ir[y].x2[i];
18         block[blockno].y1:=y;  block[blockno].y2:=y;
19         ir[y].b[i]=blockno++;
20       } // end if (!intervalmatched)
21     } // end i loop
22   } // end y loop
```

An example is shown in Fig. 2. Fig. 2(a) is the result of the execution of Algorithm 2, while the form of the array *ir*[] is also shown. The first image row (row 0) has 2 intervals, the second image row (row 1) has 4 intervals, the row before the last (row *L*-2) has no interval and the last row (row *L*-1) has 1 interval.

The execution of Algorithm 3 is described in Fig. 2 (b). The two intervals of row 0 are assigned to blocks b_0, b_1. In row 1, three new blocks b_2, b_3, b_4 are created and the interval with (x_1, x_2)=(200,358) is assigned to block b_1. Therefore, after the interval matching of the first two rows, the following blocks have been created: b_0={5,70,0,0}, b_1={200,358,0,1}, b_2={3,43,1,1}, b_3={86,126,1,1}, b_4={524,835,1,1}.

Image Row	*Number of intervals*				
0	2	(5,70,-)	(200,358,-)		
1	4	(3,43,-)	(86,126,-)	(200,358,-)	(524,835,-)
…	…				
L-2	0				
L-1	1	(501,802,-)			

(a)

Image Row	*Number of intervals*				
0	2	(5,70,0)	(200,358,1)		
1	4	(3,43,2)	(86,126,3)	(200,358,1)	(524,835,4)

(b)

Fig. 2. (a) An example of array *ir*[] after the interval extraction procedure. (b) A snapshot of the array *ir*[] after the execution of interval matching for the first two rows.

4.2 Parallel implementation of the interval extraction

Algorithm 2, for interval extraction can be easily implemented in parallel, since the image rows assigned to a thread are different from the image rows assigned to every other thread; therefore, this data partitioning scheme among the threads is suitable for parallelization. Each thread scans every image row assigned to it, finds all object level intervals in this image row and stores the results in array *ir*[].

Algorithm 4, implements the parallel interval extraction using a parallel **for** loop. The array *ir*[] is a shared variable, but each thread stores the range of image rows assigned to it. The L image rows are partitioned among the P threads using *guided* loop scheduling.

The term loop scheduling in OpenMP, refers to the type of the partitioning of the loop iterations among the threads and the optional parameter chunksize specifies the number of loop iterations assigned to a thread each time. The available loop partitioning types are the static scheduling where the loop iterations are partitioned equally among the threads and each thread is assigned L/P rows of the image, from the first row $t*L/P$ to the last row $(t+1)*L/P - 1$, where t is the thread id number; the

dynamic scheduling where the iterations are separated into chunksized bundles, each available thread receives a bundle and when is finished receives another bundle from the work queue; and the guided scheduling which is similar to the dynamic but starts with a large number of iterations and decreases in order to utilize better by avoiding threads' idling periods. As proved by the experimental data provided in Subsection 6.2, the faster execution of the Algorithm 4 is achieved using the guided loop scheduling with smallest chunk size 1, which is the default value.

Algorithm 4. *Parallel Interval Extraction*

```
1    #pragma omp parallel for schedule(guided,1) shared(ir) private(x, intervalfound)
2    for (y:=0;y<L; y++){
3      intervalfound:=0;  ir[y].irno:=0;
4      for (x:=0;  x<W-1;  x++){
5              if (img[y][x]  &&  !intervalfound){
6                intervalfound:=1;
7                ir[y].x1[ir[y].irno]:=x;
8        }
9              else if (!img[y][x]  &&  intervalfound){
10               intervalfound:=0;
11               ir[y].x2[ir[y].irno++]:=x-1;
12             }
13     } // end x loop
14     if (!img[y][W-1]) { // not last column
15       if (intervalfound) ir[y].x2[ir[y].irno++]:=W-2;
16     }
17     else { // case of last column
18       if (!intervalfound) ir[y].x1[ir[y].irno]:=W-1;
19       ir[y].x2[ir[y].irno++]:=W-1;
20     }
21   }// end y loop
```

4.3 Parallel implementation of interval matching

A parallel implementation of Algorithm 3, has to resolve the D2 and D3 data dependency issues that were previously discussed. Using the vector *block*[] and the variable *blockno* in their current form, the data race condition D2, cannot be resolved efficiently in parallel. Also, there is no efficient solution for the dependency D3, for the parallel arrangement of the indices of the vector *block*[].

A suitable data structure for the blocks is a 2D array *b* with number of rows equal to the number of threads, or equivalently a vector *b* where each element is a pointer to a vector of the extracted blocks from a specidic thread. Each block is stored in the array *b* and specifically in the *i*-th row and the *j*-th column *b*[*i*][*j*], where the indices indicate

that is the *j*-th block extracted from the *i*-th thread. The numbers of the extracted blocks are stored using the vector *bno*[], each element is the number of the extracted blocks from the corresponding thread.

There are two types of blocks for the thread *t*: The regular blocks *b*[][] that start in image rows which are assigned to the current thread *t*; thus, they belong to thread *t*. The orphan blocks *o*[][] that start in previous image row, and are continued in the rows of the current thread *t*; therefore is a list of temporary blocks that they belong to different threads. The vector *ono*[] holds the number of orphan blocks for every thread.

Algorithm 5, implements the parallel interval matching. In the following a description of the algorithm is presented. *P* is the number of threads, *L* is the number of image rows, each thread has id t = 0, 1, .., *P*-1, and initially *bno*[*t*] has the value 0. Each thread is assigned a bundle of *L/P* image rows using a static partitioning scheme; the static partitioning ensures that each thread *t* is assigned the following image rows [*t***L/P* , (*t*+1)**L/P*-1]. Each thread *t* examines and matches the intervals of its corresponding rows from the vector *ir*[] and extracts the blocks.

For the first image row *y* assigned to the thread *t*, the thread checks the intervals of *ir*[*y*] with the intervals of *ir*[*y*-1] of previous row *y*-1. In the case of no matching, then a new block is created and stored to the *b*[*t*][*bno*[*t*]] and *bno*[*t*] is increased. In the case of matching with an interval of row *y*-1 the corresponding block does not belong to thread *t,* but to a previous thread *t*-1, *t*-2, e.t.c. At this specific time the previous thread *t*-1 examines its first image rows and not its last row *y*-1, thus the intervals of row *y*-1 are not yet assigned to any regular block *b*. Therefore, the matched interval of row *y* is registered as orphan block *o*[*t*][*ono*[*t*]] by thread *t*. Fig. 3 demonstrates the process of the creation of regular and orphan blocks.

As discussed in Subsection 3.1 and presented in Fig. 2, eash interval stores the block id that is assigned to. The interval that is assigned to an orphan is designated by a negative index, in order to be distinguished from the intervals that assigned to regular blocks. In order to simplify the code, the initial value of *ono*[*t*] is 1 and increases. Then an interval *k* of row *y,* that belongs to an orphan block is assigned the followind block id, *ir*[y].*b*[*k*] = - *ono*[*t*];. Since *ono*[*t*] starts from value 1, the number of orphan blocks at this thread is *ono*[*t*]-1. At the following image rows, intervals that match with an interval of an orphan block are also assigned to the same orphan block.

After the extraction of regular and orphan blocks for each thread, there is a barrier for thread's suncronization. In the sequel each thread t checks its regular block k that exists on its last image row with the orphan block m of the next thread t+1. If there is a matching then the end row y_2 of the regular block is assigned the value of y_2 of the orphan block, i.e. $b[t][k].y2 = o[t+1][m].y2$. If the end of the orphan $o[t+1][m].y2$ is the last row of thread t+1, then the process is repeated for the orphans of thread t+2, etc., until the regular block is assinged the correct y2 value.

When the parallel region ends, the threads are joined and the structures *ir*[], *o*[][], *ono*[] are released from memory.

The structure *b*[][] uses non contiguous memory and in cases that a block has to be accessed isolated from previous block then the access time is $O(P)$. Since P is the number of threads, it has relatively small values and does not introduces any drawback.

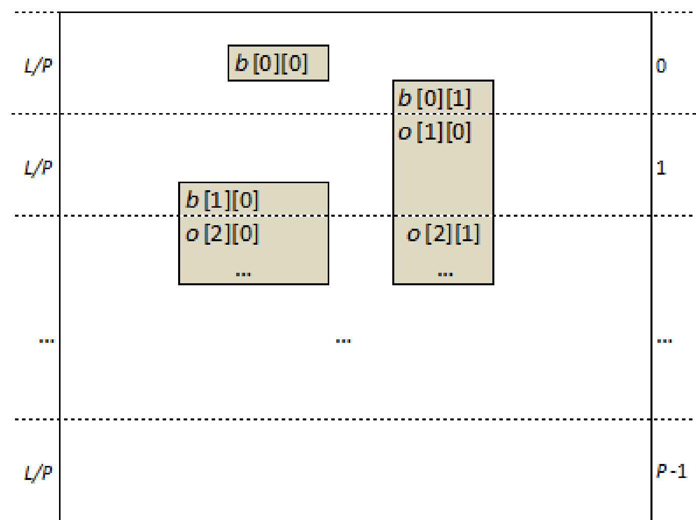


Fig. 3. The L image rows are partitioned among the P threads.The image has three blocks, *b*[0][0] which lie entirely at the rows of thread 0, *b*[0][1] starts at thread 0 and its continuation at thread 1 is initially registered as orphan *o*[1][0]; *b*[1][0] starts at thread 1 and its part at thread 2 is initially registered as orphan *o*[2][0].

Algorithm 5. *Parallel Interval Matching*

```
1   #pragma omp parallel shared(ir, b, bno,  P, first_y, M) private(intervalmatched,
    j_prev, j_curr, t, i, j, y,flag, next_t)
2   {
3     t=omp_get_thread_num();
4     P=omp_get_num_threads();
```

```
5        M = L * W /(2*P); // max blocks per thread
6        bno[t]=0; ono[t]=1;
7        flag=0;
8        #pragma omp for schedule(static)
9        for (y=0; y<L; y++) {
10         if (!flag) { first_y[t]=y; flag=1; }
11         if (y == 0)
12           for (i=0; i<ir[0].irno; i++)  {
13             b[t][bno[t]].x1=ir[y].x1[i];
14             b[t][bno[t]].x2=ir[y].x2[i];
15             b[t][bno[t]].y1=y;
16             b[t][bno[t]].y2=y;
17             ir[y].b[i]= t*M+(bno[t])++;
18           }
19         else {
20           j_prev=0;j_curr=0;
21           for (i=0; i<ir[y].irno; i++) { //for each interval i of current row y. Index j
     indicates an interval of previous row
22             intervalmatched=0;
23             for (j=j_prev; j<ir[y-1].irno && ir[y].x1[i] >= ir[y-1].x1[j]; j++)
24               if (ir[y].x1[i]==ir[y-1].x1[j] && ir[y].x2[i]==ir[y-1].x2[j]) {
25                 intervalmatched=1; j_curr=j;
26                 if (y==first_y[t]) { // 1st row for thread t, matched interval
27                   ir[y].b[i]=-ono[t];  //Intervals of orphan have negative b index
28                   o[t][ono[t]].x1=ir[y].x1[i]; o[t][ono[t]].x2=ir[y].x2[i];
29                   o[t][ono[t]].y1=y; o[t][ono[t]++].y2=y;
30                 }
31                 else { // other than 1st row of thread t, matched interval
32                   ir[y].b[i]=ir[y-1].b[j];
33                   if (ir[y].b[i] >=0) b[t][ir[y].b[i]-t*M].y2=y;
34                   else o[t][-ir[y].b[i]].y2=y;
35                 }
36             }
37             j_prev = j_curr;
38             if (! intervalmatched) { // create new block
39               b[t][bno[t]].x1=ir[y].x1[i]; b[t][bno[t]].x2=ir[y].x2[i];
40               b[t][bno[t]].y1=y; b[t][bno[t]].y2=y;
41               ir[y].b[i]=t*M+(bno[t])++;
42             }
43           }
44         }
45       }
46       #pragma omp barrier // wait  to finish all threads
47       y = (t+1)*L/P-1;  // y = last row of current thread t
48       for (i=0; i<ir[y].irno && t < P-1; i++) { // each interval of last row of thread t
49         if (ir[y].b[i] < t*M) continue; // only blocks that belong to thread t
50         next_t=t+1;
51         do { // check b[ir[y].b[i]] with the orphan blocks of next threads
52           flag=1;
53           for (j=1; j<ono[next_t]; j++) {
```

```
54          if (b[t][ir[y].b[i]-t*M].x1 == o[next_t][j].x1 && b[t][ir[y].b[i]-t*M].x2 ==
    o[next_t][j].x2) {
55            b[t][ir[y].b[i]-t*M].y2 = o[next_t][j].y2;
56            if ((next_t < P-1) && (o[next_t][j].y2 == first_y[next_t+1] - 1)) {
57              flag = 0; next_t++;
58            }
59            break;
60          }
61        }
62      } while (! flag);
63    }
64  } // parallel region of interval matching
```

5 Theoretical analysis of the PIBR algorithm

The PIBR algorithm consists of the parallel Algorithm 4 for interval extraction and the parallel Algorithm 5 for interval matching. We define t_1, t_2, t_3, as the execution times of the serial Algorithms 1, 2, 3; $t_4(P)$ and $t_5(P)$ as the execution times of parallel Algorithms 4, 5, respectively, using $P \geq 1$ cores, and $t_{PIBR}(P)$ as the execution time of the PIBR algorithm using P cores, where $t_{PIBR}(P) = t_4(P) + t_5(P)$.

A variety of performance metrics of parallel processing have been used [22]-[27]. According to the well known Amdahl's law [22], the maximum achievable speedup S is defined as:

$$S = \frac{t_{SIBR}}{t_{PIBR}} = \frac{1}{f + \frac{1-f}{P}} \tag{2}$$

where t_{SIBR} and t_{PIBR} are the serial and parallel execution times, respectively, P is the number of the processors used and $0 \leq f \leq 1$ is the fraction of the work that is executed sequentially; thus $1-f$ is the parallel fraction of the work.

Gustafson [23] suggested the scaled speedup and according to the known as Gustafson-Barsis's law the maximum achievable speedup is:

$$S = P + (1-P)f \tag{3}$$

Amdahl's and Gustafson-Barsis's laws they do not take into account the parallel overhead and therefore overestimate the speedup.

The parallel overhead [25] incurs from the parallelization process, the idling time of the processors caused by imbalanced workload and the communication among processors when using shared variables in case of OpenMP. The parallel overhead is in general a function of the problem size w and the number of processors P. Taking

into account the parallel overhead t_o and considering the PIBR algorithm, speedup is written as:

$$S(w,P)=\frac{t_2(w)+t_3(w)}{\frac{t_2(w)+t_3(w)}{P}+t_o(w,P)} \tag{4}$$

The problem size is defined in terms of the total number of basic operations of the algorithm. It is beyond any doubt that the execution time of the IBR process depends on the image size, i.e. the number of image pixels and on the content of the input image [8], [9]. The qualitative characteristics that affect the execution time of the IBR for a given image size, is the density and the patterns appeared in the image.

In the Appendix, a latency analysis is presented considering the estimation of the clock cycles of the interval extraction of an image with size WxL, that contains m intervals. The analysis concludes that the total required clock cycles of the serial Algorithm 2 is:

$$A\lg 2_lat \simeq 14WL+25m \tag{5}$$

Therefore, the problem size w_2 of Algorithm 2 is analogous to Alg2_*lat* and the execution time $t_2=\Theta(14WL+25m)$, where $\Theta(x)$ is a function with the same growth rate as x. Since the number of pixels WL is usually much greater than the number m of the intervals, it is concluded without loss of generality that the term $14WL$ grows much faster than the term $25m$. Therefore, the main factor that affects the serial interval extraction time is the image size, while the image density and content is a secondary factor.

The problem size for the interval matching process, depends on the number L of image rows, on the number m of the intervals and also on the number k of the blocks which is related to the distribution of the intervals among the image rows.

In our analysis and the experiments, we use some real life images as representative average cases and also some extreme image cases. The best image case for IBR is the zero image $f(x,y)=0,\forall x,y$, since no intervals and blocks are contained. The worst image case for IBR is the chessboard image, since it can reach the maximum number Lx$W/2$ of intervals and blocks, if the square size is (1x1) pixel.

5.1 Estimation of the execution time of PIBR Algorithm

Consider the subproblem of finding the object level intervals in a binary image, where the binary image is located in memory and the object level intervals are stored in

memory using the data structure of Fig. 2 (a). Then, the Algorithm 2 is an effective serial solution and the parallel Algorithm 4 is an effective parallel solution to this problem. The same applies for the subproblem of interval matching.

Examining the pairs of serial Algorithm 2 with parallel Algorithm 4 and serial Algorithm 3 with parallel Algorithm 5, it is concluded that the serial Algorithms are fully parallelizable and there is no serial fraction in interval extraction and interval matching. From parallel Algorithm 4, it is obvious that the processing of any image row is independent of the processing of any other image row and there is no communication among the threads using shared variables. Therefore, it is difficult to derive a theoretical estimation of the parallel overhead. A similar conclusion applies for the parallel Algorithm 5.

The parallel overhead acts like a serial fraction e of work that existed in parallel execution, caused mainly by the imbalanced workload, thread synchronization and other sources of overhead, such as, the architectural overhead. The Karp-Flatt metric [25] for the experimentally determined serial fractions e_4, e_5 of Algorithms 4 and 5, respectively, is defined as:

$$e_4(w_2,P)=\frac{1/S(w_2,P)-1/P}{1-1/P} \tag{6a}$$

$$e_5(w_3,P)=\frac{1/S(w_3,P)-1/P}{1-1/P} \tag{6b}$$

where w_2, w_3 are the problem sizes of Algorithm 2, 3, where $t_2=\Theta(w_2)$ and $t_3=\Theta(w_3)$.

Profiling the Algorithms 1, 2, 3, 4 and 5 for zero images of different sizes and different number of cores used, the execution times t_{Z1}, t_{Z2}, t_{Z3}, $t_{Z4}(P)$ and $t_{Z5}(P)$ are obtained. From these values the speedup values $S_{Z4}(w_{Z2},P)$, $S_{Z5}(w_{Z3},P)$ and the experimentally determined serial fractions $e_{Z4}(w_{Z2},P)$, $e_{Z5}(w_{Z3},P)$ of Algorithms 4 and 5 for Zero images are calculated.

For any other input image, if the problem sizes w_2, w_3 are known, or equivalently if the serial times t_2, t_3 are known, it is feasible to estimate the parallel interval extraction and parallel interval matching times, by making the assumption that the values of the experimentally determined serial fractions e_{Z4}, e_{Z5} of zero image are quite similar to the corresponding values $e_4(w_2,P)$, $e_5(w_3,P)$ of the input image with same size, i.e., $e_4(w_2,P)\simeq e_{Z4}(w_2,P)$, $e_5(w_3,P)\simeq e_{Z5}(w_3,P)$. This assumption is quite

reasonable taking into account the fact that the important factor that affects t_2 and t_3 is the image size, while the number of intervals is a secondary factor as denoted by the latency analysis. Therofore, the estimated PIBR execution time for any given input image is defined as:

$$\hat{t}_{PIBR}(P) = \frac{t_2 + t_3}{P} + t_o(P) = \frac{t_2 + t_3}{P} + t_2 \cdot e_{Z4}(P) + t_3 \cdot e_{Z5}(P) \quad (7)$$

5.2 Speedup and Efficiency metrics

The relative speedup S_R [26] is defined as the ratio of the execution time of the parallel algorithm using 1 processor to the execution time of the parallel algorithm using P processors.

$$S_R = \frac{t_{PIBR}(1)}{t_{PIBR}(P)} \quad (8)$$

The absolute speedup S_A [27] is defined as the ratio of the execution time of the best sequential algorithm to the execution time of the parallel algorithm using P processors.

$$S_A = \frac{t_{SIBR}}{t_{PIBR}(P)} \quad (9)$$

In (6a), (6b) the absolute speedup values are used and for P=1 the denominators are zero and the serial fractions e_4 and e_5 are undefined. The overhead time $t_o(P)$ and the estimated PIBR time $\hat{t}_{PIBR}(P)$ are also undefined for P=1.

The efficiency metric is defined as the ratio of speedup to P, $E = S / P$. They are defined both absolute and relative efficiency metrics, depending on the speedup values that were used.

5.3 Isoefficiency and scalability analysis

The scalability of a parallel system is the ability to increase performance as the number of the processors increases. A scalable system should increase speedup in such a rate that the efficiency is maintained as the number of processors increases. However, the parallel overhead increases as the number of processors increases and the efficiency decreases as the number of processors increases. Increasing the problem size is a way to maintain efficiency. The isoefficiency [27], [28] analysis investigates the rate of problem size increase with respect to the number of processors in order to

maintain a constant efficiency. Obviouly, the isoefficiency is related to the scalability of a parallel system.

The total amount of overhead $T_o(w,P)$ is the time spent by all processors carrying out work not done by the sequential algorithm:

$$T_o(w,P) = Pt_o(w,P) = P\left[t_2 \cdot e_4(w,P) + t_3 \cdot e_5(w,P)\right] \tag{10}$$

Also, the total overhead time is interpreted as $T_o(w,P) = P \cdot t_{PIBR} - t_{SIBR}$. Then, the maximum achievable speedup in (2), is rewritten as:

$$\begin{aligned} S(w,P) &\leq \frac{t_2+t_3}{\frac{t_2+t_3}{P}+t_o(w,P)} = \frac{P\left(t_2+t_3\right)}{t_2+t_3+Pt_o(w,P)} \Rightarrow \\ S(w,P) &\leq \frac{Pt_{SIBR}}{t_{SIBR}+T_0(w,P)} \end{aligned} \tag{11}$$

The efficiency is the ratio of speedup to P:

$$E(w,P) \leq \frac{t_{SIBR}}{t_{SIBR}+T_0(w,P)} \Rightarrow t_{SIBR} \geq \frac{E(w,P)}{1-E(w,P)} T_0(w,P) \tag{12}$$

If a constant efficiency is maintained as the number of processors increases, the fraction $C=E/(1-E)$ in (12) is also a constant. In the analysis of Subsection 5.1, it has been assumed that the experimentally determined serial fractions of the zero image e_{Z4}, e_{Z5} are equal to the corresponding values e_4, e_5 of the input image, and since $t_0 = t_2 e_{Z4} + t_3 e_{Z5}$, (12) is rewritten as:

$$t_2 + t_3 \geq CP(t_2 e_{Z4} + t_3 e_{z5}) \tag{13}$$

This last relation is the isoefficiency relation and in order to maintain a constant efficiency as the number of the processors increases, the problem size t_2+t_3 should be increased so that the above inequality is satisfied. Solving for P, (13) is rewritten as:

$$P \leq \frac{t_2+t_3}{C[t_2 e_{Z4} + t_3 e_{z5}]} \tag{14}$$

This relation gives the number of processors required in order to maintain a constant efficiency. It is concluded from (13) and (14), that the scalability of the PIBR Algorithm depends on the input image and the serial execution times t_2 and t_3.

6 Experimental results

For the experimental evaluation, a platform consisting of DELL PowerEdge R820 nodes, with four Intel(R) Xeon(R) CPUs E5-4650v2 which is based on SandyBridge

EP microarchitecture, with nominal frequency 2.4GHz, each with 40 cores and 512 GB RAM, of the National HPC facility ARIS, of the Greek Research & Technology Network (GRNET), was utilized. The operating system is Centos 6.7 Linux. All the programs were implemented in C using the OpenMP API and were compiled using the Intel compiler icc ver. 15.0.3.

In order to compare the sequential and the parallel algorithms, the execution time was used as a measure. The computation times are measured from the execution of the IBR algorithms only, excluding image reading from disk, etc. The execution time starts when the relative function of IBR is called, with parameters, a pointer to the address of the image in memory, the image width W and the image length L, and ends when the function returns the number of the blocks *bno*[] and the blocks *b*[][].

All time complexities were measured using the omp_get_wtime() function of OpenMP. To decrease random variation, all the execution time complexities were measured as the average of 1000 runs.

(a) (b) (c)

Fig. 3. A set of test images: (a) Shapes, (b) the negative of the image Page, (c) Chess-10. All the test images were used in different sizes from (1024x1024) to (30000x30000) pixels.

Several sets of test images were used to evaluate the performance of the parallel algorithms, with varying sizes from (1024x1024) to (30000×30000). In Fig. 3 three samples of test images are presented. The test image Shapes, of Fig. 3(a), has small number of transitions from 0 to 1, since there are large homogeneous areas; such conditions often appear in industrial imaging applications, such as manufacturing and robot vision. The test image Page, of Fig. 3(b), is the negative of the image of a book

page with printed characters and there is a significantly greater number of transitions from 0 to 1 with smaller homogeneous areas; these conditions appear in document processing and character recognition applications. The test image Chess-10, of Fig 3 (c), with square size of (10x10) pixel. The Chess-10 test image does not relate to any vision applications and has been used in order to test the proposed algorithms. The zero image has been used as a test image, while, the Chess-1, a chessboard image with square size of (1x1) pixel was used. The Chess-1 image is the worst case image for IBR since it reaches the maximum number of blocks Lx$W/2$ where L,W are the image sizes. The zero image is the simplest image case for IBR, since no block is extracted.

A large number of experiments has been conducted, on different algorithms and their variations for the parallelization of the IBR, using different images with different sizes and number of cores, from 1 to max 40. Actually, more than 55000 Core hours on the ARIS HPC platform have been consumed on computations.

6.1 Compiler optimization levels

The experimental measurements have been repeated for three different compiler optimization levels: O0 which means no compiler optimization; O2 which produces executable code optimized for speed by enabling parallelization and vectorization; and O3 which includes O2 optimization plus some more aggressive loop and memory access optimizations, such as loop unrolling, IF statement collapsing and is suggested in applications that have loops with many floating point operations or process large data sets [29].

Table 1 demonstrates the time complexities of the Algorithms 1, 2 and 3, using the compiler optimization levels O0, O2, O3. Since the O2 compiler optimization level has the better results, all the following experimental results are measured from executables that compiled using the "-O2 -xCORE-AVX-I" compiler optimization flags, where the flag -xCORE-AVX-I produces Intel(R) Xeon(R) E5-v2 processor specific code.

6.2 Loop partitioning type of Algorithm 4

As discussed in Subsection 4.2, the loop scheduling type in Algorithm 4 is *guided* with the default chunk size value 1. All the available scheduling types, with different chunk sizes, for p threads from 1 to 40 and for 1000 runs have been used in experiments. The chunk size in static schedule is L/P, where L is the number of

image rows. The dynamic scheduling was evaluated for the following 5 chunksizes: 1, $L/4P$, $2L/4P$, $3L/4P$, L/P. The chunk size parameter in the guided schedule specifies the smallest chunk size, and was evaluated for the values 1, $L/8P$, $L/4P$. The execution times are demonstrated in Table 2; it is observed that the guided outperforms the other scheduling types, due to better utilization of the processors. Moreover, the smallest chunk size 1 has the better results. Similar results have been obtained for all number of threads, therefore the guided scheduling with chunk size of 1 efficiently handles the load balance among the threads and for this reason is used in the parallel Algorithm 4.

6.3 Time complexities and performance

Table 3 presents the experimental data for the Zero images of different sizes. The first (upper) part of Table 3 presents the execution times of the serial Algorithms 1, 2, 3; the second part presents the execution time $t_4(P)$, the absolute speedup S_{A4} and the Karp-Flatt metric of experimentally determined serial fraction e_{Z4} of Algorithm 4; the third part presents the same data of Algorithm 5; and finally the fourth part presents the execution time $t_{PIBR}(P)$ and its relative speedup S_R and efficiency E_R values. Table 3 and the subsequent Tables present results only for a specific number of cores (powers of 2 and the maximum 40) in order to preserve the readability.

Tables 4, 5 and 6, present the estimated and the experimental values of time complexities and performance metrics for the test images Shapes, Page and Chess-10 of different sizes. The first part of each Table presents the number of intervals and blocks and the execution times of the serial Algorithms 1, 2 and 3. The second part of each Table presents the estimated and the experimentally measured execution times of PIBR Algorithm. The third part of each Table presents the relative speedup S_R and efficiency E_R values of the PIBR Algorithm. A number of interesting observations arise from Tables 3 - 6.

1. The execution times t_2 and t_3 of the serial interval extraction depends on the input image; this observation validates the analysis of the execution cost of Algorithm 2 which is based on instruction latency and presented in the Appendix.

2. Significant speedup values of 32.58, 33.60, 30.38, 19.00 achieved for images Zero, Shapes, Page and Chess-10 respectively using 40 cores have been obtained. Also, it is observed that the efficiency values increase as the image size increases and decrease as the number of cores increases.

3.The experimental and theoretically estimated times t_{PIBR}, $\hat{t}_{\mathrm{PIBR}}$ of PIBR Algorithm are quite close for all the test images and these results validate the theoretical analysis. The relative speedup values for the test images of different sizes are presented graphically in Fig. 4.

6.4 Scalability

The scalability depends on the input image as demonstrated by the theoretical analysis and by the experimental data. Table 7 demonstrates the number of cores and the image size required in order to maintain relative efficiency at value 0.80, each row presents the real and estimated number of cores for a test image at different sizes. The real values received from the experimental data and the estimated values using eq. (14) for C=4, which corresponds to E=0.80. Since (14) involves P, $e_{Z4}(P)$ and $e_{Z5}(P)$ cannot be solved analytically, therefore a recursive approach from P=1 to 40 is used until the maximum value of P that satisfies (14) is found, for any given image and size. From these results it is observed that for the average case test images the PIBR algorithm is scalable, but the PIBR algorithm has lower scalability for the Chess-10 images. These results are presented graphically in Fig. 5. Thus, it is concluded that the scalability of PIBR Algorithm depends on the input image.

6.5 Memory bandwidth issues

The memory system bandwidth affects negatively high performance applications, leaving the processors idle as they wait for memory. This is known as memory wall and is a well known problem in parallel processing [32].

The granularity problem [33]-[35] mainly causes the memory bandwidth problem. Also, the numerous occurring memory reads/writes trigger the cache coherence effect leading to a memory wall. When these reads/writes are distributed in depth in different cache lines [36] the whole effect and performance results are normalized.

In order to determine if the presented parallel algorithms are affected by the memory bandwidth, a different image example is used. The Chess-1 image is a chessboard with square size of (1x1) pixel and is the extreme case for IBR since it contains the maximum number of intervals and blocks. In this image case although the granularity size is not degenerated, the transition from 0 to 1 luminance values occurs in every other pixel, therefore there is an interval to store every two-pixel values read. In Table

8, are presented the execution times of parallel interval extraction and the evaluated minimum memory bandwidth due to pixel reads and interval writes (excluded several other overheads, i.e., the read/write of variable *intervalfound*, addressing, instruction fetch, etc.).

It is observed that due to the memory bandwidth problem for Chess-1 test images and sizes from (4Kx4K) and greater, the fastest execution takes place for 8 cores. The parallel interval matching algorithm is not affected and the speedup increases with the number of cores. For the maximum image size of (30000x30000) pixels less than 1GB of memory required to fit, for the maximum number of blocks less than 8GB of memory required to fit; since each node has 512GB of RAM there is no memory amount limitation in the PIBR algorithm and the test images used.

7 Conclusions and Future work

In this paper, the parallelization of a sequential algorithm for the block representation of binary images has been investigated. The sequential algorithm is not directly parallelized due to data dependencies and has been decomposed into two discrete parts and the corresponding algorithms. The two parts of interval extraction and interval matching have been parallelized effectively using the OpenMP API and the proposed parallel algorithm (PIBR) achieved significant speedup values.

The complexity of the proposed PIBR algorithm depends on the input image and specifically to the image size and the content of the image. A theoretical analysis and estimation of the parallel algorithm execution times and the corresponding performance metrics is not directly feasible. The Karp-Flatt metric of experimentally determined serial fraction, was used in the Zero test image in order to measure the parallel overhead of the interval extraction and the interval matching tasks. The values of the Karp-Flatt metric were used for the estimation of the execution times of PIBR Algorithm for other input images and the predicted values were very close to the real ones and this validates our analysis. A scalability analysis and the isoefficiency relation has also been given.

In the Appendix, a latency analysis is given for the total required clock cycles for the interval extraction process, as a function of the number of image pixels WL and the number m of the extracted intervals. A similar analysis concerning the interval

matching process is not possible, since it depends on the number m of the intervals and on the distribution of the m intervals among the L image rows.

The representation of binary images with blocks allows concurrent machine perception of greater image areas than a pixel; this approach proved superior, as compared to the 2D array image representation. This feature has been used in the past for the performance enhancement of various image processing sequential algorithms on von Neumann computers. There are many ideas and directions for future work as the parallel implementation of the sequential algorithms presented in [8]-[13] on block represented images and more computational time gains are expected. Also, the parallel implementation of the IBR algorithm and the operations on block represented images, presented in [8]-[13] using CUDA on GPGPUs, is another quite interesting direction for our future work.

8 Acknowledgments

This work was supported by computational time granted from the Greek Research & Technology Network (GRNET) in the National HPC facility - ARIS - under project ID PA170601-PIBR. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that have improved the presentation of our work.

Appendix. Instruction latency analysis of Algorithm 2

The clock cycle period t_c is considered as the latency of the execution of the simplest instruction, such as the execution of the instruction *'AND R1 R2'*. According to Intel(R) [30]: "Due to the complexity of dynamic execution and out-of-order nature of the execution core, the instruction latency data may not be sufficient to accurately predict realistic performance of actual code sequences based on adding instruction latency data". There is a difficulty to derive the real execution time from the total clock cycles periods and the reason is that today's processors are quite complex, they decode and use micro-operations (or μops), they execute multiple instructions per clock cycle (IPC) and they also use advanced hardware optimizations such as pipelining and branch prediction. However, an instruction latency analysis gives an indication of the complexity of an algorithm and an estimation of the execution time. In order to estimate the execution cost of each operation of the Algorithm 2, the

Intel(R) microarchitecture with codename Sandy Bridge that used on the processors of the experiments was considered [31].

For the zero image the loop of variable y at Line 1 requires L increments and L conditional jumps each with cost $1t_c$ and a total latency of $2Lt_c$. In Algorithm 2, in the loop of Line 3 the loop control variable x increases up to W-2. In Lines 13-19, a similar process is repeated for the last column of the image, therefore without loss of generality it is assumed that the loop at Line 3 considered that has W repetitions. The loop of variable x at Line 3 requires WL increments and WL conditional jumps and a total of $2WL$ clock cycles. The movement of a pixel value from memory to register depends on the data locality to memory hierarchy and a reasonable requirement is $4t_c$ and for all image pixels a minimum total of $4WLt_c$. The execution of each if statement at Lines 4 and 8, requires the execution of two comparisons on the pixel value and the flag *intervalfound* in registers, the execution of one logical AND between registers and the execution of the conditional branch of the if statement each with cost $1t_c$, therefore a total of $4WLt_c$ for each of the two if statements. Since every pixel of zero image has 0 value, the flag *intervalfound* is always 0 and the conditions of the two if statements in Lines 4 and 8 are always false. This also means that both ***if*** statements are always executed with a total of 8 WLt_c. Moreover, L movements from register to memory of the *ir*[].*irno* values with a minimum total of $4Lt_c$ are required. Therefore, it is concluded that a total of $14WLt_c+6Lt_c$ is required for the zero image.

In other images that contain a number m of intervals, there are m interval starting pixels and m interval ending pixels. For the interval starting pixels the condition of the 1st ***if*** statement is true and therefore the execution of the first if statement with $4mt_c$, the m settings of the flag *intervalfound* with mt_c, the m settings of the coordinate x_1 with mt_c and m unconditional jumps to return to the next pixel checking with mt_c are required . The second ***if*** statement at Line 8 is not executed. Moreover, m movements from register to memory of the values of x_1 with a minimum total of $4mt_c$ are required. Therefore, for interval starting pixels, a total of $11mt_c$ is required for the two if statements.

For the m interval ending pixels, the two if statements are executed with $8mt_c$ and also the condition of the second if statement is true for interval ending pixels. Therefore the two variable settings *intervalfound* and x_2 and the increment of *ir*[].*irno* are executed and also m unconditional jumps to return to the next pixel checking with a total $4mt_c$ are required for the execution of the branch of the 2nd if statement.

Moreover, m movements from register to memory of the values of x_2 with a minimum total of $4mt_c$ are required and the total time units for the interval endings is $16mt_c$.

Since $2m$ pixels of the m intervals require different latency times, the term $2mt_c$ should be subtracted from the term $14WLt_c+6Lt_c$ of the zero images. Therefore, the instruction latency of interval extraction for a WxL image that contains m number of intervals using the Algorithm 2 is:

$$A\lg 2_lat = 14WL + 6L - 2m + 11m + 16m = 14WL + 6L + 25m \Rightarrow \qquad (16)$$
$$A\lg 2_lat \simeq 14WL + 25m$$

According to the above relation, the significance of the image size to the execution time of Algorithm 2 is much greater than the significance of the number of the extracted intervals, since usually $WL >> m$. Even in the case of the Chess-10 image with size (30000x30000), which contains $9 \cdot 10^8$ pixels and $45 \cdot 10^6$ intervals, the term $14WL$ is 11.2 times greater than the term $25m$, while in Chess-1 image with size (30000x30000), the term $14WL$ is 1.1 times greater than the term $25m$.

References

[1] Samet H. (1984) The quadtree and related hierarchical data structures. Computing Survey 16:187-260

[2] Capon J. (1959) A propabilistic model for run-length coding of pictures. *IRE* Trans. Information Theory IT-5:157-163

[3] Pratt W.K. (1991) Digital Image Processing 3rd edn. John Wiley & Sons, New York

[4] Piper J. (1985) Efficient implementation of skeletonisation using interval coding. Pattern Recognition Letters 3:389-397

[5] Freeman H. (1974) Computer processing of line drawings. ACM Computing Surveys 6:57-97

[6] Paglieroni D.W., Jain A.K. (1988) Control point transfroms for shape representation and measurement. Computer Vision, Graphics and Image Processing 42:87-111

[7] Kashyap R.L., Chellappa R. (1981) Stochastic models for closed boundary analysis: Representation and reconstruction. IEEE Trans. Information Theory, IT-27:627-637

[8] Spiliotis I., Mertzios B. (1996) Fast algorithms for basic processing and analysis operations on block representated binary images. Pattern Recognition Letters 17:1437-1450

[9] Spiliotis I., Mertzios B. (1998) Real-time computation of two-dimensional moments on binary images using image block representation. IEEE Trans Image Processing 7:1609-1615

[10] Spiliotis I., Mertzios B. (1997) A Fast Parallel Skeletonization Algorithm on Block Represented Binary Images. Elektrik 1:161-173

[11] Spiliotis I., Mertzios B. (1997) A Fast Skeleton Algorithm on Block Represented Binary Images. In: 13th International Conference on Digital Signal Processing (DSP97)

[12] Gatos B., Perantonis S., Papamarkos N. (1996) Accelerated Hough Transform using rectangular block decomposition. Electronic Letters 32:730-732

[13] Spiliotis I., Boutalis Y. (2011) Parameterized real-time moment computation on gray images using block techniques. Journal of Real-Time Image Processing 6:81-91

[14] R. Chandra, L. Dagum, D. Kohr , D. Maydan , J. McDonald, R. Menon (2001) Parallel Programming in OpenMP, Academic Press, USA

[15] Chatzidoukas P.E., Dimakopoulos V.V., Delakis M., Carcia C. (2009) A high-performance face detection system using OpenMP. Concurrency Comput: Pract.Exper. 21:1819-1837

[16] Rasmussen M., Stuart M., Karlsson S., (2009) Parallelism and Scalability in an Image Processing Application. Int J Parallel Prog 37:306–323

[17] Mahmoudi R., Akil M., Hedi Bedoui M. (2017) Concurrent computation of topological watershed on shared memory parallel machines. Parallel Computing 69 : 78–97

[18] Szwoch G., Ellwart D., Czyzewski A. (2016) Parallel implementation of background subtraction algorithms for real-time video processing on a supercomputer platform. Journal of Real-Time Image Proc 11:111–125

[19] Cabaret L., Lacassagne L., Etiemble D. (2016) Parallel Light Speed Labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors. Journal of Real-Time Image Processing, DOI 10.1007/s11554-016-0574-2

[20] Hosny K. et al. (2017) Fast computation of 2D and 3D Legendre moments using multi-core CPUs and GPU parallel architectures. Journal of Real-Time Image Processing, DOI 10.1007/s11554-017-0708-1

[21] Lu Y. et al., (2016) Parallelizing image feature extraction algorithms on multi-core platforms. Journal of Parallel and Distributed Computing 92: 1–14

[22] G.M. Amdahl (1967) Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press: 483-485

[23] J. L. Gustafson (1988) Reevaluating Amdahl's Law. Comm. ACM 31 : 532-533

[25] Karp A.H., Flatt H.P. (1990) Measuring Parallel Processor Performance. Comm ACM 33 : 539-543

[26] X-H. Sun and J. L. Gustafson (1991) Toward a better parallel performance metric. Parallel Computing 17 : 1093-1109

[27] A.Y. Grama, A. Gupta, V. Kumar (1993) Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. IEEE Parallel Distrib. Systems 1: 12-21

[28] Quinn M.J. (2003) Parallel Programming in C with MPI and OpenMP. McGraw-Hill.

[29] Intel Corporation (2017) Step by Step Performance Optimization with Intel® C++ Compiler, https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler,

[30] Intel Corporation (2016) Intel® 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32- architectures-optimization-manual.pdf

[31] Fog A. (2018) Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. https://www.agner.org/optimize

[32] Hutcheson A., Natoli V. (2011) Memory Bound vs . Compute Bound : A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications. Stone Ridge Technology, Internal White Paper.

[33] ChenJack S., Dongarra J., Hsiung C. (1984) Multiprocessing linear algebra algorithms on the CRAY X-MP-2: Experiences with small granularity. Journal of Parallel and Distributed Computing 1, 22-31.

[34] Chen D.-K., et al (1990) The Impact of Synchronization and Granularity on Parallel Systems. Proc. 17th Annual Intern. Symposium Computer Architecture, Seattle, Washington, USA.

[35] Gentile A.,Sander S., Wills L., Wills S., (2004) The impact of grain size on the efficiency of embedded SIMD image processing architectures. Journal of Parallel and Distributed Computing 64: 1318-1327.

[36] Intel Corporation (2011) Avoiding and Identifying False Sharing Among Threads. https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

[37] Cisco inc. (2017) Visual Networking Index: Forecast and Trends, 2017–2022 White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

Table 1. The time complexities t_1, t_2, t_3 of Algorithms 1, 2, 3 expressed in msec, using different compiler optimization levels, for the test images of size (30000x30000).

Image Size	Zero 30000x30000			Shapes 30000x30000			Page 30000x30000			c10 30000x30000		
Opt level	O0	O2	O3	O0	O2	O3	O0	O2	O3	O0	O2	O3
t_1	3289.08	942.89	949.23	3285.56	1088.36	1089.34	3347.42	1028.78	1031.12	4383.37	2145.81	2200.24
t_2	3063.28	1271.07	1286.11	3048.70	1399.74	1399.81	3073.11	1340.93	1336.96	3398.19	2367.19	2346.37
t_3	0.092	0.082	0.079	7.74	4.69	4.70	27.74	14.28	13.52	1046.41	241.29	244.77

Table 2. The time complexities of Algorithm 4 expressed in msec, using different types and chunksize of loop scheduling, for test images of different sizes. L is the number of image rows and P=40 is the number of threads. The results are similar for all the image sizes used.

Loop Scheduling	Chunksize	Shapes 4Kx4K	Shapes 16Kx16K	Page 4Kx4K	Page 16Kx16K	Chess-10 4Kx4K	Chess-10 16Kx16K
Static	L/P	1.43	23.38	1.40	22.61	1.70	29.24
Dynamic	1	0.90	13.74	1.89	19.31	7.18	112.66
Dynamic	$L/4P$	0.87	14.31	0.98	18.78	1.47	23.99
Dynamic	$2L/4P$	1.08	20.93	1.12	18.69	1.71	28.30
Dynamic	$3L/4P$	1.12	18.73	1.23	17.43	2.00	30.77
Dynamic	L/P	1.31	24.39	1.41	23.22	1.65	27.81
Guided	Smallest 1	0.76	11.94	0.82	13.24	1.51	22.23
Guided	Smallest $L/8P$	0.92	13.09	0.93	14.62	1.56	22.98
Guided	Smallest $L/4P$	0.81	13.59	0.87	13.37	1.52	22.48

Table 3. First (upper) part:The execution times of Algorithms 1, 2, 3; second part: The execution time t_4, the absolute speedup S_{A4} and the experimentally determined serial fraction e_{Z4} of the parallel Algorithm 4; third part: The execution time t_5, the absolute speedup S_{A5} and the experimentally determined serial fraction e_{Z5} of the parallel Algorithm 5; fourth part: The execution time t_{PIBR}, the relative speedup S_R and the relative efficiency E_R of the PIBR algorithm using 1 to 40 cores for the Zero images of different sizes. All the time complexities are the average of 1000 runs and are expressed in msec.

Image size	1024x1024			1920x1080			2048x2048			4096x4096			8192x8192			16384x16384			30000x30000		
t_1	1.12			2.2			4.41			17.51			69.99			279.8			942.9		
t_2	1.48			2.93			5.89			23.37			93.21			378.18			1271.07		
t_3	0.001			0.002			0.003			0.005			0.017			0.040			0.082		
Cores	t_4	S_{A4}	e_{Z4}	t_4	S_{A4}	e_{Z4}	t_4	S_{A4}	e_{Z4}	t_4	S_{A4}	e_{Z4}	t_4	S_{A4}	e_{Z4}	t_4	S_{A4}	e_{Z4}	t_4	S_{A4}	e_{Z4}
1	1.49	0.99		2.96	0.99		5.97	0.99		23.40	1.00		93.27	1.00		379.02	1.00		1248.63	1.02	
2	0.75	1.97	0.014	1.50	1.95	0.024	2.96	1.99	0.005	11.74	1.99	0.005	47.46	1.96	0.018	192.56	1.96	0.018	631.21	2.01	-0.007
4	0.38	3.89	0.009	0.75	3.91	0.008	1.50	3.93	0.006	5.92	3.95	0.004	23.58	3.95	0.004	96.08	3.94	0.005	318.53	3.99	0.001
8	0.20	7.40	0.012	0.40	7.33	0.013	0.78	7.55	0.008	3.07	7.61	0.007	12.30	7.58	0.008	51.04	7.41	0.011	166.17	7.65	0.007
16	0.12	12.33	0.020	0.22	13.32	0.013	0.41	14.37	0.008	1.60	14.61	0.006	6.29	14.82	0.005	26.40	14.33	0.008	86.42	14.71	0.006
32	0.09	16.44	0.031	0.13	22.54	0.014	0.23	25.61	0.008	0.82	28.50	0.004	3.17	29.40	0.003	13.57	27.87	0.005	43.57	29.17	0.003
40	0.08	18.50	0.030	0.12	24.42	0.016	0.20	29.45	0.009	0.69	33.87	0.005	2.66	35.04	0.004	12.26	30.85	0.008	38.29	33.20	0.005
Cores	t_5	S_{A5}	e_{Z5}	t_5	S_{A5}	e_{Z5}	t_5	S_{A5}	e_{Z5}	t_5	S_{A5}	e_{Z5}	t_5	S_{A5}	e_{Z5}	t_5	S_{A5}	e_{Z5}	t_5	S_{A5}	e_{Z5}
1	0.003	0.33		0.005	0.40		0.008	0.38		0.015	0.33		0.039	0.44		0.086	0.47		0.154	0.53	
2	0.003	0.33	5.00	0.005	0.40	4.00	0.006	0.50	3.00	0.011	0.45	3.40	0.028	0.61	2.29	0.064	0.63	2.20	0.109	0.75	1.66
4	0.002	0.50	2.33	0.004	0.50	2.33	0.005	0.60	1.89	0.008	0.63	1.80	0.018	0.94	1.08	0.034	1.18	0.80	0.063	1.30	0.69
8	0.002	0.50	2.14	0.004	0.50	2.14	0.004	0.75	1.38	0.006	0.83	1.23	0.017	1.00	1.00	0.029	1.38	0.69	0.051	1.61	0.57
16	0.001	1.00	1.00	0.003	0.67	1.53	0.003	1.00	1.00	0.005	1.00	1.00	0.014	1.21	0.81	0.026	1.54	0.63	0.046	1.78	0.53
32	0.001	1.00	1.00	0.003	0.67	1.52	0.002	1.50	0.66	0.004	1.25	0.79	0.012	1.42	0.70	0.022	1.82	0.54	0.043	1.91	0.51
40	0.001	1.00	1.00	0.002	1.00	1.00	0.002	1.50	0.66	0.004	1.25	0.79	0.010	1.70	0.58	0.020	2.00	0.49	0.039	2.10	0.46
Cores	t_{PIBR}	S_R	E_R	t_{PIBR}	S_R	E_R	t_{PIBR}	S_R	E_R	t_{PIBR}	S_R	E_R	t_{PIBR}	S_R	E_R	t_{PIBR}	S_R	E_R	t_{PIBR}	S_R	E_R
1	1.49	1.00	1.00	2.97	1.00	1.00	5.98	1.00	1.00	23.42	1.00	1.00	93.31	1.00	1.00	379.11	1.00	1.00	1248.78	1.00	1.00
2	0.75	1.98	0.99	1.51	1.97	0.99	2.97	2.02	1.01	11.75	1.99	1.00	47.49	1.96	0.98	192.62	1.97	0.98	631.32	1.98	0.99
4	0.38	3.91	0.98	0.75	3.93	0.98	1.51	3.97	0.99	5.93	3.95	0.99	23.60	3.95	0.99	96.11	3.94	0.99	318.59	3.92	0.98
8	0.20	7.39	0.92	0.40	7.34	0.92	0.78	7.63	0.95	3.08	7.61	0.95	12.32	7.58	0.95	51.07	7.42	0.93	166.22	7.51	0.94
16	0.12	12.34	0.77	0.22	13.30	0.83	0.41	14.47	0.90	1.61	14.59	0.91	6.30	14.80	0.93	26.43	14.35	0.90	86.47	14.44	0.90
32	0.09	16.41	0.51	0.13	22.29	0.70	0.23	25.77	0.81	0.82	28.42	0.89	3.18	29.32	0.92	13.59	27.89	0.87	43.61	28.63	0.89
40	0.08	18.43	0.46	0.12	24.30	0.61	0.20	29.59	0.74	0.69	33.74	0.84	2.67	34.95	0.87	12.28	30.87	0.77	38.33	32.58	0.81

Table 4. First part: The number of intervals, the number of blocks, the execution times of the Algororithms 1,2,3; second part: The predicted values $\hat{t}_{PIBR}$ and the real values t_{PIBR} of execution times of PIBR Algorithm; third part: The relative speedup and efficiency of PIBR Algorithm for the Shapes images of different sizes. All the time complexities are expressed in msec and the real execution times are the average of 1000 runs.

Image size	1024x1024		1920x1080		2048x2048		4096x4096		8192x8192		16384x16384		30000x30000	
Intervals	2,078		2,735		8,062		16,348		33,399		67,975		128,264	
Blocks	1,672		2,302		7,041		11,247		16,547		22,942		36,207	
t_1	1.38		2.69		5.34		20.83		`		326.23		1088.36	
t_2	1.71		3.33		6.72		26.50		105.09		416.73		1399.74	
t_3	0.035		0.043		0.165		0.469		1.138		2.290		4.693	
Cores	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$
1		1.89		3.52		7.02		27.54		109.06		422.32		1436.64
2	1.07	0.92	1.94	1.76	3.97	3.56	15.21	13.97	57.65	54.75	222.20	213.47	700.47	732.63
4	0.53	0.47	0.97	0.90	2.07	1.85	7.70	7.09	28.20	27.26	108.84	107.34	355.47	361.94
8	0.31	0.25	0.56	0.49	1.15	1.01	4.14	3.79	15.25	14.37	58.69	56.51	187.39	189.78
16	0.18	0.15	0.32	0.27	0.65	0.56	2.32	2.11	8.12	7.47	30.87	29.34	98.47	99.14
32	0.14	0.11	0.22	0.17	0.38	0.31	1.32	1.16	4.41	3.88	16.31	15.00	50.65	50.53
40	0.13	0.11	0.18	0.15	0.34	0.28	1.17	0.99	3.69	3.25	14.76	12.72	44.64	42.80
Cores	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	2.06	1.03	2.01	1.00	1.97	0.99	1.97	0.99	1.99	1.00	1.98	0.99	1.96	0.98
4	4.04	1.01	3.90	0.97	3.80	0.95	3.89	0.97	4.00	1.00	3.93	0.98	3.97	0.99
8	7.51	0.94	7.23	0.90	6.94	0.87	7.26	0.91	7.59	0.95	7.47	0.93	7.57	0.95
16	12.79	0.80	13.08	0.82	12.65	0.79	13.05	0.82	14.60	0.91	14.39	0.90	14.49	0.91
32	17.69	0.55	20.34	0.64	22.72	0.71	23.84	0.75	28.13	0.88	28.16	0.88	28.43	0.89
40	17.86	0.45	23.15	0.58	25.44	0.64	27.73	0.69	33.60	0.84	33.20	0.83	33.57	0.84

Table 5. First part: The number of intervals, the number of blocks, the execution times of the Algororithms 1,2,3; second part: The predicted values $\hat{t}_{PIBR}$ and the real values t_{PIBR} of execution times of PIBR Algorithm; third part: The relative speedup and efficiency of PIBR Algorithm for the Page images of different sizes. All the time complexities are expressed in msec and the real execution times are the average of 1000 runs.

Image size	1024x1024		1920x1080		2048x2048		4096x4096		8192x8192		16384x16384		30000x30000	
Intervals	28,601		30,706		57,202		114,404		228,808		457,616		838,326	
Blocks	18,753		22,092		21,563		26,385		27,972		29,972		119,471	
t1	2.23		3.55		6.77		23.43		83.75		430.36		1028.78	
t2	2.22		3.92		7.70		28.10		105.07		599.15		1340.93	
t3	0.410		0.375		0.699		1.720		3.470		6.762		14.283	
Cores	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}
1		3.15		4.77		8.82		31.34		111.88		427.78		1388.82
2	3.40	1.97	3.74	2.70	6.34	4.88	20.89	17.72	64.16	58.07	328.83	216.76	692.17	701.45
4	1.63	1.26	1.98	1.66	3.47	2.79	10.68	9.33	31.29	30.88	160.13	111.26	349.75	352.30
8	1.23	0.92	1.39	1.03	2.08	1.81	6.05	5.27	17.87	17.27	87.20	58.46	186.30	187.06
16	0.62	0.68	0.90	0.69	1.28	1.15	3.76	3.53	10.16	9.85	46.78	31.44	100.15	99.20
32	0.56	0.43	0.76	0.47	0.78	0.81	2.41	2.15	6.11	5.82	25.42	17.17	53.81	51.55
40	0.54	0.41	0.55	0.43	0.74	0.76	2.24	1.87	5.10	5.27	23.00	15.64	47.53	45.72
Cores	SR	ER	SR	ER	SR	ER	SR	ER	SR	ER	SR	ER	SR	ER
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.60	0.80	1.76	0.88	1.81	0.90	1.77	0.88	1.93	0.96	1.97	0.99	1.98	0.99
4	2.49	0.62	2.87	0.72	3.16	0.79	3.36	0.84	3.62	0.91	3.85	0.96	3.94	0.99
8	3.41	0.43	4.63	0.58	4.88	0.61	5.94	0.74	6.48	0.81	7.32	0.91	7.42	0.93
16	4.62	0.29	6.88	0.43	7.69	0.48	8.87	0.55	11.36	0.71	13.60	0.85	14.00	0.88
32	7.40	0.23	10.21	0.32	10.88	0.34	14.57	0.46	19.24	0.60	24.92	0.78	26.94	0.84
40	7.73	0.19	11.01	0.28	11.55	0.29	16.81	0.42	21.21	0.53	27.36	0.68	30.38	0.76

Table 6. First part: The number of intervals, the number of blocks, the execution times of the Algororithms 1,2,3; second part: The predicted values $\hat{t}_{PIBR}$ and the real values t_{PIBR} of execution times of PIBR Algorithm; third part: The relative speedup and efficiency of PIBR Algorithm for the Chess-10 images of different sizes. All the time complexities are expressed in msec and the real execution times are the average of 1000 runs.

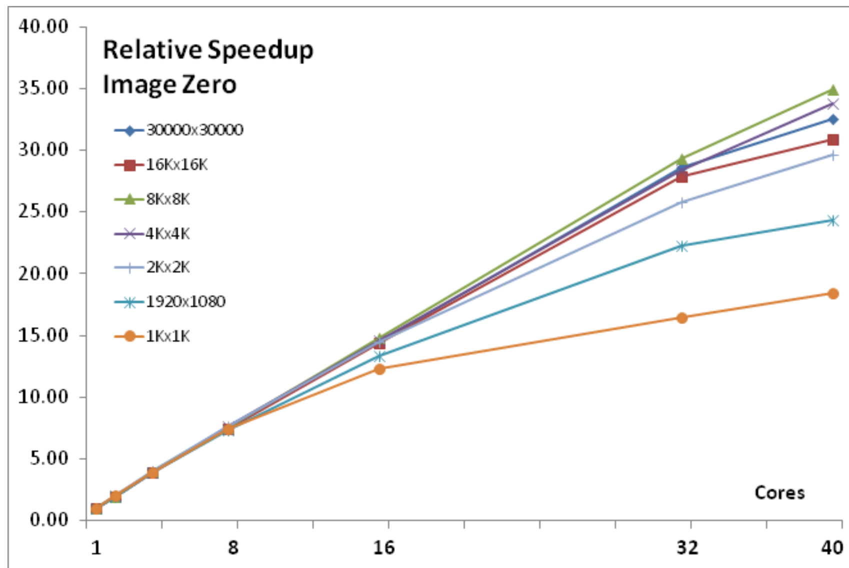
Image size	1024x1024		1920x1080		2048x2048		4096x4096		8192x8192		16384x16384		30000x30000	
Intervals	52,734		103,584		209,916		839,680		3,358,720		13,426,686		45,000,000	
Blocks	5,304		10,368		21,012		84,050		336,200		1,343,160		4,500,000	
t_1	2.66		4.95		9.91		41.37		162.72		633.43		2145.81	
t_2	2.74		5.44		10.98		43.90		174.80		688.31		2367.19	
t_3	0.288		0.569		1.391		6.070		19.877		74.220		241.287	
Cores	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}	$\hat{t}_{PIBR}$	t_{PIBR}
1		3.84		7.26		13.68		53.36		220.95		956.85		3168.01
2	2.99	1.87	5.41	3.97	10.41	7.51	45.83	30.76	146.15	117.42	557.18	526.26	1688.31	1715.82
4	1.35	1.07	2.44	2.32	4.80	4.18	18.58	16.92	49.26	67.67	214.58	286.72	507.44	912.45
8	0.98	0.72	1.83	1.39	3.06	2.45	11.50	10.63	34.83	41.80	134.47	178.31	321.95	524.11
16	0.51	0.49	1.21	0.86	2.00	1.46	8.21	6.29	23.85	26.40	89.75	98.00	226.86	289.63
32	0.45	0.35	1.07	0.55	1.26	1.01	5.92	3.90	17.73	15.67	61.97	57.20	172.58	178.33
40	0.44	0.32	0.77	0.51	1.23	0.94	5.77	3.64	14.83	14.67	56.54	51.27	157.84	166.74
Cores	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R	S_R	E_R
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	2.05	1.02	1.83	0.91	1.82	0.91	1.73	0.87	1.88	0.94	1.82	0.91	1.85	0.92
4	3.58	0.90	3.14	0.78	3.27	0.82	3.15	0.79	3.27	0.82	3.34	0.83	3.47	0.87
8	5.35	0.67	5.24	0.65	5.59	0.70	5.02	0.63	5.29	0.66	5.37	0.67	6.04	0.76
16	7.90	0.49	8.42	0.53	9.36	0.59	8.49	0.53	8.37	0.52	9.76	0.61	10.94	0.68
32	11.10	0.35	13.12	0.41	13.50	0.42	13.67	0.43	14.10	0.44	16.73	0.52	17.77	0.56
40	11.85	0.30	14.12	0.35	14.60	0.36	14.68	0.37	15.06	0.38	18.66	0.47	19.00	0.48

Table 7. The number of cores and the image size required in order to maintain efficiency at value 0.80 for the test images of different sizes. Each row presents the number of cores for a test image at different sizes.

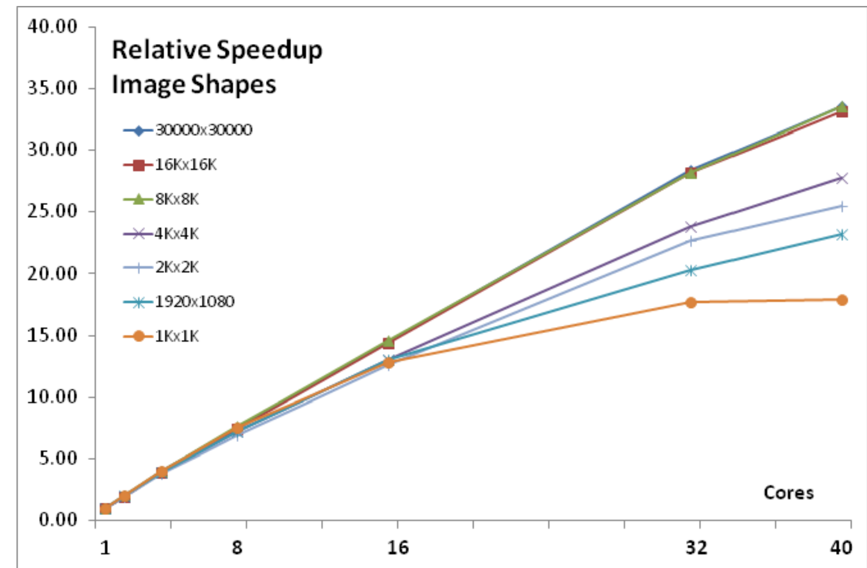
	1Kx1K		1920x1080		2Kx2K		4Kx4K		8Kx8K		16Kx16K		30000x30000	
Image	Cores Real	Cores Predicted	Cores Real	Cores Predicted	Cores Real	Cores Predicted	Cores Real	Cores Predicted	Cores Real	Cores Predicted	Cores Real	Cores Predicted	Cores Real	Cores Predicted
Zero	14	14	22	20	30	28	40	>40	40	>40	38	38	40	>40
Shapes	16	8	13	4	18	6	21	9	40	20	40	34	40	>40
Page	2	1	3	1	4	1	6	1	7	7	28	22	36	28
Chess-10	6	1	3	1	3	1	4	1	4	1	4	3	6	4

Table 8. First part: The image size, the number of pixels in millions, the number of intervals in millions, the read transfer from memory in MB for the image pixels, the write transfer to memory in MB for the intervals. Second part: The execution times of Algorithm 4 in msec for the Chess-1 images of different sizes; for sizes from 4Kx4K and greater the minimum memory bandwidth in GB/s.

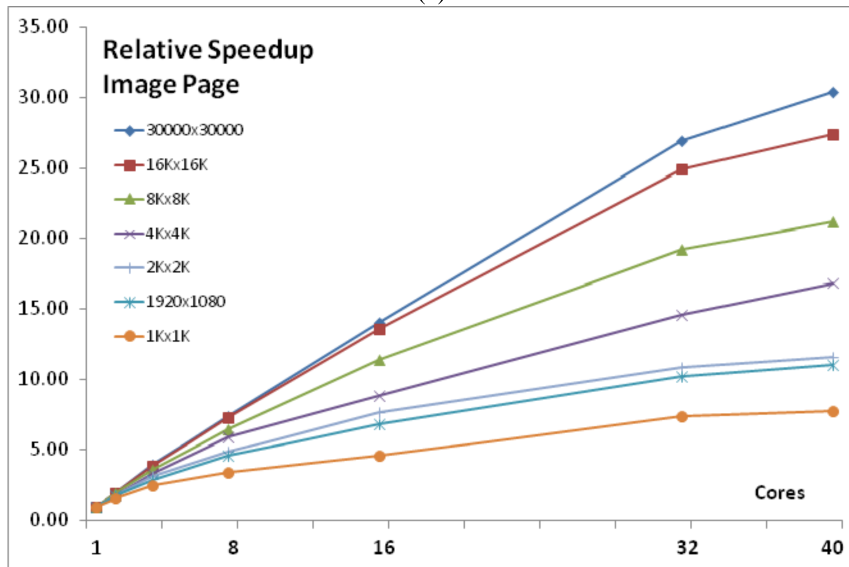
LxW	1024x1024	1920x1080	2048x2048	4096x4096		8192x8192		16384x16384		30000x30000	
Pixels (M)	1.05	2.07	4.19	16.78		67.11		268.44		900.00	
Intervals (M)	0.52	1.04	2.10	8.39		33.55		134.22		450.00	
Read (MB)	1.05	2.07	4.19	16.78		67.11		268.44		900.00	
Write (MB)	4.19	8.29	16.78	67.11		268.44		1,073.74		3,600.00	
Cores	t4 (ms)	t4 (ms)	t4 (ms)	t4 (ms)	Bandwidth (GB/s)	t4 (ms)	Bandwidth (GB/s)	t4 (ms)	Bandwidth (GB/s)	t4 (ms)	Bandwidth (GB/s)
1	2.33	4.57	9.35	45.10	1.86	173.52	1.93	646.65	2.08	2156.13	2.09
2	1.39	2.77	5.04	21.19	3.96	82.73	4.06	328.61	4.08	1094.78	4.11
4	0.81	1.60	2.57	10.63	7.89	41.43	8.10	163.75	8.20	536.53	8.39
8	0.47	0.89	1.51	5.75	14.59	34.22	9.81	119.13	11.27	379.26	11.87
16	0.33	0.53	0.97	8.39	9.99	101.28	3.31	588.56	2.28	1921.21	2.34
32	0.27	0.41	0.84	7.37	11.39	138.71	2.42	724.14	1.85	3012.83	1.49
40	0.25	0.38	0.81	6.06	13.84	142.80	2.35	795.09	1.69	2983.45	1.51



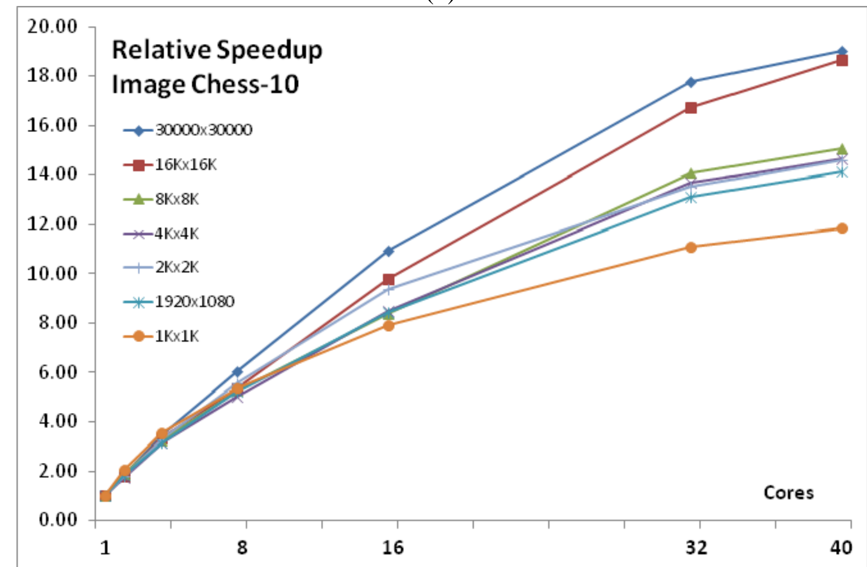
(a)



(b)



(c)



(d)

Fig. 4. The relative speedup from the execution of PIBR algorithm using up to 40 cores for (a) Zero, (b) Shapes, (c) Page and (d) Chess-10 images of different sizes.

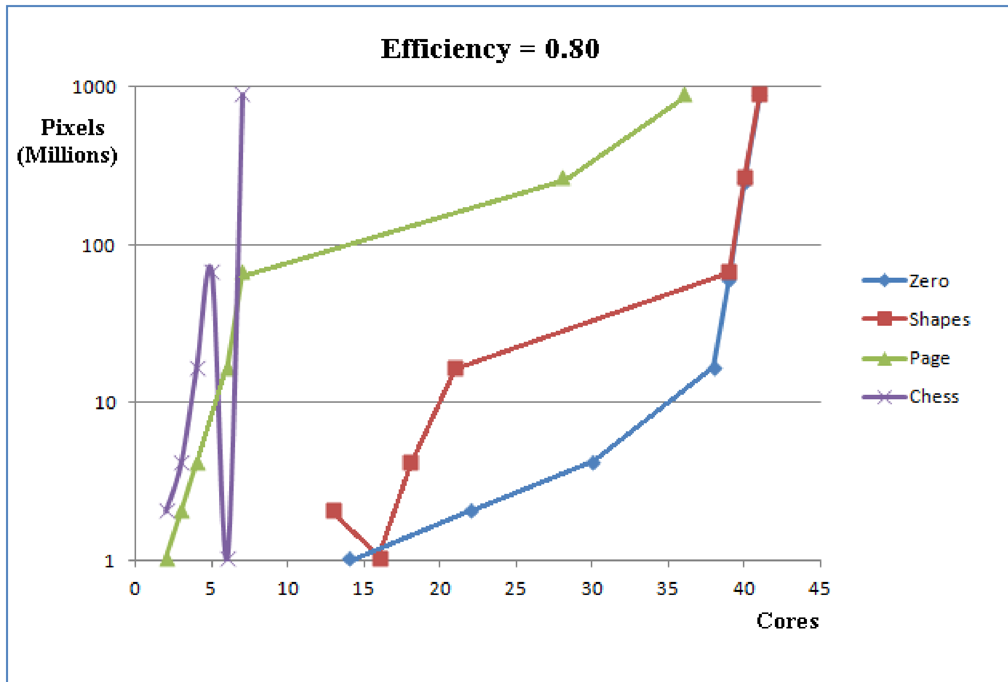


Fig. 5. The number of cores required in order to maintain efficiency of the PIBR algorithm at level 0.80 for the test images of different sizes.