MVA '96 IAPR Workshop on Machine Vision Applications, November. 12-14, 1996, Tokyo, Japan

Efficient Implementation of Image Processing Algorithms on Linear Processor Arrays using the Data Parallel Language 1DC

Sholin Kyo *
Information Technology Research Laboratory
NEC Corporation

Kan Sato †
NEC Informatec Systems, Ltd.

Abstract

SIMD linear processor arrays (LPAs) have received a great deal of interest as a suitable parallel architecture for image processing. However, few possess a high level programming environment support, and the range of image processing tasks which can be efficiently implemented is unclear. In this paper, we first describe a data parallel language succinctly designed for a virtual LPA, and also a compiler for an existing LPA. Next, we provide a guideline for parallel SIMD linear array algorithm development using the language. The guideline is consisted of five basic parallelizing methods, by using which efficient implementations are shown for each category of low to intermediate level image operations. We also suggest that further improvement of performance on LPAs can be achieved, by architectural supports for reducing the control overhead of some parallelizing methods.

1 Introduction

SIMD linear processor arrays (LPAs) have received a great deal of interest as a suitable parallel architecture for image processing [1]–[3]. However, when focusing on the software environment, few possess a high level programming language support. Although some parallel image processing algorithms have been proposed thus far[9]-[11], currently there is a lack of a clear idea to what extent can parallelism be exploited for image tasks by using LPAs.

In this paper, 1DC (One Dimensional C), a succinctly defined data parallel language which supports a virtual LPA, and a 1DC compiler developed for an existing LPA, IMAP-VISION[4], are first described. Then, for the categories of low to intermediate level image operations, a guideline for their parallel SIMD linear array algorithm developement using 1DC is provided. The guideline is consisted of five basic parallelizing methods: *row, column, row-systolic, slant-systolic*, and *stack-based*. Furthermore, overheads for some of the parallelizing methods are discussed and compared; based on which future subjects for further improving the performance of LPAs for a wider variety of image processing tasks are suggested.

*Address: 4-1-1 Miyazaki, Miyamae-ku, Kawasaki, 216, Japan. E-mail: sholin@pat.cl.nec.co.jp

†Address: Kanagawa Science Park 3-2-1 Sakado, Takatu-ku, Kawasaki, 213, Japan. E-mail: k-sato@ats.nis.nec.co.jp

2 1DC Language Features and its compiler for IMAP-VISION

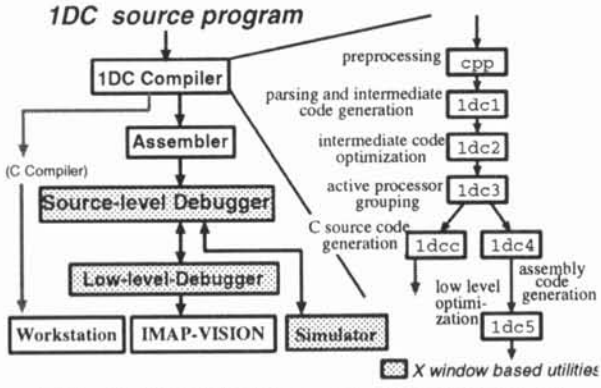
1DC is designed as an enhanced C language, with the enhancement limited only to essential necessities for the sake of clarity, and also for the support of a virtual LPA (which is described in the beginning of section 3). The enhancement of 1DC from C is straightforward: (a) extended declaration of entities which associated to the PE array, (b) extended constructs for selecting active processor groups, and (c) extended operators for manipulating data on the PE array.

Entities are declared as either *sep* (or *separate*) and associated to the PE array, or *scalar* and is associated to the central controller. Each *sep* entity represents a linear array of scalar data where each element of which resides on the corresponding PE. Extended constructs for selecting active PE groups partition the PEs into two sets, where the first set is composed by PEs that verify the predicate of the construct, and the second set is composed by all other remaining PEs. These constructs are given a preceding *m* for their notation such as *mif...[melse...]*, *mwhile...*, and *mfor(...;...;...)*, where the only difference from standard C is the predicate must be a *sep* expression.

In the same way, extended operators are given a preceding colon for their notation. Assuming c0,···,cn as constants, E_{sep} and E_{sca} respectively as a *sep* and a *scalar* expression, :(c0,···,cn :) represents a *sep* constant with value c0,···,cn on the $0th$,···,nth PE counting from the leftmost in a LPA; "E_{sep}:[E_{sca} :]" extracts the scalar element of E_{sep} on the $E_{sca}th$ PE; ":>E_{sep}" and ":<E_{sep}" respectively refers to the scalar element of E_{sep} located at its left and right adjacent PEs; finally ":&&E_{sep}" and ":||E_{sep}" respectively produces a *scalar* entity whose value is the logical AND and OR of every scalar element of E_{sep}.

Currently a 1DC compiler has been developed for

IMAP-VISION[4], a highly integrated single-board LPA with 256 PEs. Fig.1(a) shows the current programming environment for IMAP-VISION based on 1DC. Due to the succinct language design and the RISC like instruction-set of IMAP-VISION, the 1DC compiler has achieved codes competitive with hand-written assembly codes (Fig.1(b)). The compiler can also produce C source codes for running 1DC on native PCs and workstations. The X window based debuggers provide not only both assembly and source level debugging facilities, but also provide functionalities such as interactive variable adjustment which is useful for parameter tuning in real-time image applications.



(a)The IMAP-VISION programming environment

application	assembly code steps (A)	compiler code steps (B)	ratio (B/A)	time of (B) in msec
binarization	1547	2402	1.55	0.061
average filter	5600	6430	1.15	0.161
histogram	4039	4872	1.21	0.122
90 deg. rotation	20696	23326	1.13	0.583

(Image size 256x256, using IMAP-VISION in 40MHz)

(b) Some evaluation results

Figure 1: 1DC Programming Environment

3 Efficient Implementation of Image Processing on LPAs

Low and intermediate level image operations can be classified into some categories (Fig.2, partly based on [5]). In this section, we provide a guideline for parallel SIMD linear array algorithm development using 1DC.

Our target machine is a virtual LPA (Fig.3). The source and destination image sizes are both NROW×NCOL where NCOL is equal to the number of PE (PENO). Thus, each image column is mapped onto a different PE, and is stored in the PE's local memory. All PEs are controlled by a central controller, which performs instruction broadcast, sequential access to any address of the local memory of any PE, and also status information reduction (receives the logical OR or AND of status signals from all PEs). Each PE can perform access to different local memory address (indirect addressing facility). Interconnections exists only between adjacent PEs, while the leftmost PE is connected to the rightmost PE.

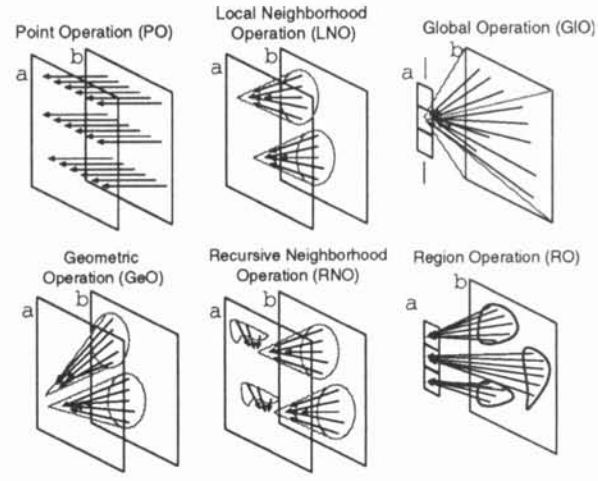


Figure 2: Low and intermediate level image processing categories

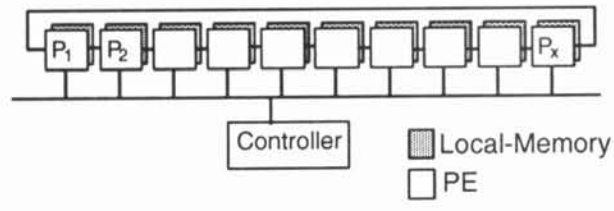


Figure 3: Architecture of a virtual LPA

3.1 Point Operation (PO) and Local Neighbourhood Operation (LNO)

Both PO and LNO are the basic parallel pixel(s)-to-pixel transformation process between source and destination images. The straightforward parallel implementation for PO and LNO on LPAs is to operate on each image row (NCOL pixels) simultaneously by all PEs, and is repeated NROW times. Hereafter this basic method is referred to as *row* method.

The 1DC description for the 3×3 average filtering operation (a typical LNO), is given in the following. The sum of each local 3×3 pixels are obtained by combining the sum of three 1×3 pixels produced by each PE with the result of its two adjacent PEs.

```
sep unsigned char src[NROW],dst[NROW];
void average_filter()
{
 sep unsigned int   acc;
 int   i;
```

```
  for(i=1; i<NROW-1; i++){
     acc = src[i-1] + src[i] + src[i+1];
     acc += (:<acc + :>acc);
     dst[i] = (acc / 9);
  }
}
```

3.2 Global Operation (GlO) and Geometrical Operation (GeO)

GlO is mainly used for gathering information from pixels in order to produce a single value or a vector of values as the result, while GeO is mainly used for repositioning pixels. On LPAs, both GlO and GeO can be efficiently achieved by first letting all PEs simultaneously perform vertical and then horizontal data transfer, or vice versa. Vertical data transfer can be achieved by using the indirect addressing mechanism which enables each PE simultaneously access a pixel value in each different row. Horizontal data transfer can be achieved by utilizing the PE interconnection to simultaneously *shift* up to NCOL data in a fixed (left or right) direction. By using 1DC, vertical data transfer is expressed by using *sep* entities as array indexes, and horizontal data transfer is expressed by using a *sep* entity as the source and as well the destination operand of the :> or :< operator in a loop. Hereafter the former is referred to as *column* method, and the later is referred to as *row-systolic* method.

In the following, the 1DC description for the image histogram calculation (a typical GlO), is given as an example for implementing GlOs on LPAs. The original algorithm can be found in some where like in [9]. First, based on the *column* method, each PE generates in its local memory (a column-wise histogram array), whose starting address differs in a regular way according to each PE number. Next, these column-wise histogram arrays are summed up in a right to left direction based on the *row-systolic* method, by which after PENO iterations, the number of pixels whose grey value is equal to g. That is, the histogram result for grey value g where $0 \leq g \leq 255$, is obtained on the PE whose PE number is g. Fig. 4 illustrates briefly the above summing sequence, using a LPA with only 4 PEs and a 4×4 sized source image for brevity. Note that in the following 1DC description, PENUM is a pre-defined *sep* constant equal to :(1,2,···, PENO :). The performance of this 1DC description is 0.12 *msec*, as shown in Fig. 1(b).

```
sep unsigned char src[NROW],hst[256];
sep unsigned int histogram()
{
 sep unsigned int result=0;

 /* column-wise local histogram generation */
 for(i=0;i<NROW;i++)  hst[(src[i] - PENUM)&255]++;

 /* summation of column-wise histogram results */
 for(i=0;i<NCOL;i++)  result= :<(hst[i%256] + result);
 return(result);
}
```

The 1DC description for the 90 degree rotation (a typical GeO), based on the use of the combination of *column method* and *row-systolic method*, is shown

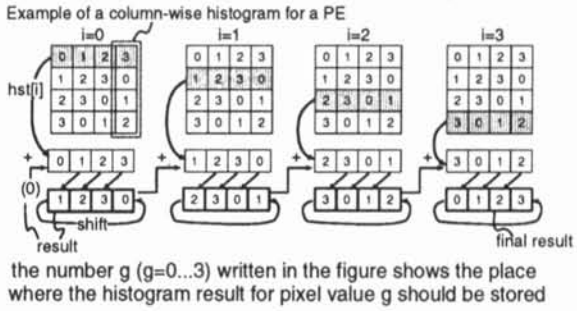


Figure 4: Row systolic operations on column-wise histograms for a 4x4 image on a LPA with 4 PEs.

in the following (the original algorithm is from [10]). By performing consecutively the following 1) ~ 3), the source image is rotated 90 degree as shown in Fig. 5: 1) a PE number dependent vertical shift of each image column (based on the *column* method), 2) a row number dependent horizontal shift of each image row (based on the *row-systolic* method), and finally 3) a PE number dependent vertical shift of each image column. The performance of this 1DC description is 0.58 *msec*, as shown in Fig. 1(b).

```
void rotate90(src,tbl)
sep unsigned char src[],tbl[];
{
 int          i;

 /* vertical shift */
 for(i=0;i<NROW;i++) tbl[(i-PENUM)&255]= src[i];

 /* horizontal shift */
 for(i=0;i<NROW;i++) tbl[i] = tbl[i] :< (PENO-i);

 /* vertical shift */
 for(i=0;i<NROW;i++) src[i] = tbl[(i+PENUM+1)&255];
}
```

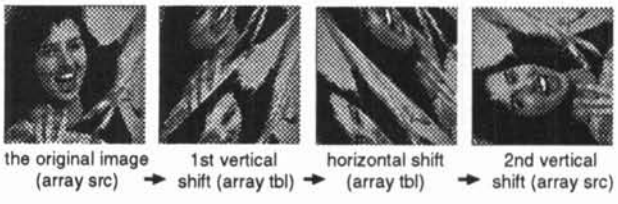


Figure 5: 90 degree rotation of a sample image

3.3 Recursive Neighbourhood Operation (RNO)

For updating each pixel, RNOs refer the pixel value of its neighbourhood pixels which have already been updated. In consequence, for RNOs constraints exist in pixel updating order between each pixel and its neighbourhood pixels. However, as the constraints are in most cases static, they are expressible as a recursive mask such as those shown in Fig. 6(a). Among Fig. 6(a), A and B are the typical recursive masks for respectively left-top to right-bottom and right-bottom to left-top raster scan operation, the most frequently used RNOs.

A parallelizing method called *slant-systolic* is proposed here for efficiently implementing RNOs on LPAs, In *slant-systolic* method, PEs are activated successively in a fixed direction, while each activated PE updates the corresponding pixel after every fixed time interval. As a result, a slant pixel-updating-wave is generated, whose slant angle is in proportion to the fixed time interval. For $N \geq 2$, it takes $N\times(NROW-1)+NCOL$ iterations for a $(2N-1)\times N$ sized recursive mask, and $N\times(NCOL-1)+NROW$ iterations for a $N\times(2N-1)$ sized recursive mask, to proceed the pixel-updating-wave from one corner to the opposite corner of the image (Fig.6(b)). Note that the fixed time intervals are both N-1.

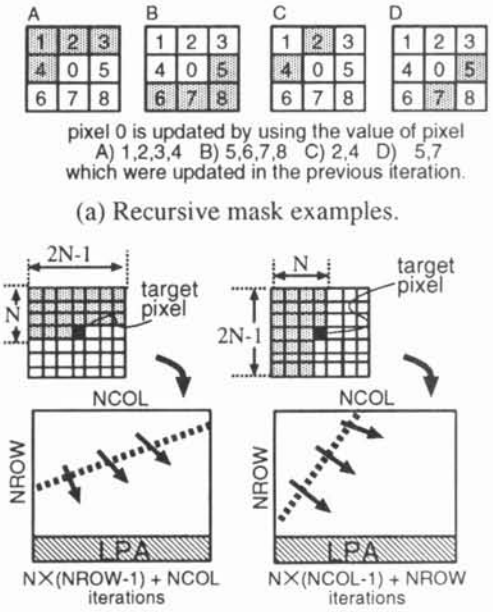
The 1DC description for the forward scan of the two-scan distance transform[6], a RNO which uses the recursive mask A and B (Fig. 6(a)), for the forward and the backward scan, is shown in the following. Two *sep* entities, s and y are used, where s is for propagating an activation signal from the leftmost PE to the rightmost PE. Each PE to which the signal has arrived, starts to increase y in every iteration. However, the PE updates the pixel at position $y/2$ only when y is an even number in order to observe the time interval imposed by the recursive mask A (for the above case the time interval is one, as N equals to two).

```
sep unsigned char img[NROW]; /* source image */

#define D 3  /* 8-nbh distance */
#define S 2  /* 4-nbh distance */
#define min(a,b)  (((a)>(b))? (b) : (a))

void dt(sep unsigned char y, sep unsigned char in[])
{
  sep unsigned char p1,p2,p3,p4,p5;

  p1= :>in[:<y-1]; p2= in[y-1]; p3= :<in[:>y-1];
  p4= :>in[:<y];   p5= in[y];
  return min(min(min(p5,p1+D),min(p2+S,p3+D)),p4+S);
}

void slant-systolic-method()
{
  int i;
  sep int  s,y;

  for (y=0,i=0; i< 2*(NROW-1)+NCOL; i++) {
    s:[0:] = 1;
    mif (s && (y++ &1)==0)
       dt(y>>1,img);
    s = :>s;
  }
}
```

Note that, due to the prescribed time interval, for RNOs using a $(2N-1)\times(2N-1)$ sized recursive mask, up to N successive pixel-updating-waves can be implemented in an overlapped way (Fig.6(c)) with some minor modification of the above 1DC program including preparing N sets of s and y.

3.4 Region Operation (RO)

One of the frequently used procedure in image processing is segmentation. After performing segmentation, usually various regions with arbitrary sizes and shapes are found within the image. RO can be used for visiting some or all pixels of each region independently, in order to produce a vector of results whose elements each of which corresponds to



(a) Recursive mask examples.

(b) Row (left) and column direction (right) pixel-updating-waves.

(c) Overlapping pixel-updating waves

Figure 6: Implementation aspects of the *slant-systolic* method for RNOs

each region. The feature of RO is that, unlike other image operation categories, source pixels are now scattered and located within specific regions, each being separated by non-source pixels. Furthermore, pixels within a region may be updated in parallel (PO, LNO, GeO, or GlO within regions: *parallel RO*), or constraints may exist in the updating order of each pixel in the region (RNO within regions: *sequential RO*), or even pixels which need to be updated may change dynamically (RO within regions: *dynamic RO*). Examples for *parallel RO* are erosion, dilation, relaxation schemes (such as Snakes[8]). Examples for *sequential RO* are contour tracing, distance transform, and a typical example for *dynamic RO* is skeletonization or thinning. Note that, *dynamic RO* is not further discussed in this paper due to space limitation.

Usually ROs are considered as intermediate level image operation, and have not been efficiently implemented on LPAs thus far. Instead, the idea of

parallelizing the implementation of RO has been to use a SIMD-MIMD hierarchy architecture, and assign the operation for each region to each MIMD processor. However, we propose in the following a parallelizing technique called *stack-based* method, by which efficient implementation of ROs on LPAs can be achieved to a large extent.

The *stack-based* method is consisted of two processing phases during each of which every PE of the LPA simulates a software stack in its local memory. In the first processing phase (the *seed pixel detection* phase), all pixels are visited once by *row* method in order to find at least one specific feature pixels (such as contour or peak point pixels) for each region, and push each pointer of the feature pixel into the stack top of the PE which possess the pixel in its local memory. In the second processing phase (the *push and pop* phase), 1) for each PE whose stack is not empty, pop the pixel pointer at the stack top and perform the RO specific operation upon the pixel pointed by the pointer (the *focused pixel* hence); 2) for each neighbourhood pixel of the *focused pixel* which satisfies the RO specific condition (the *push condition* hence), push its pointer to the stack top of the PE which possesses it in its local memory; 3) continue 1) ~ 2) until all PE stacks are empty.

The 1DC description for the above procedures 1) ~ 3) are shown in the following, where `IsSeed()`, `Pixel_op()`, and `Push_nbh_ptrs()` are RO specific functions.

```
void stack-based_method(sep int stack[], sep int img[])
{
  int i;
  sep unsigned char sp,x,IsSeed();
  void pixel_op(), push_nbh_pixels();

  for (i=0; i<NROW; i++) /* seed pixel detection */
      mif (IsSeed(i)) stack[sp++]=i;

  mwhile (:||sp) {        /* push and pop */
      x = stack[--sp];
      Pixel_op(x,img);
      Push_nbh_ptrs(x);
  }
}
```

By providing proper *push condition* for each RO, only pixels belonging to the same region as the *focused pixel*, and together pixels which really need to be processed, are identified and pushed into PE stacks, and thus being processed. As a result, for *parallel ROs*, the maximum number of pixels which has to be processed by a PE, and for *sequential ROs* such as contour tracing, the maximum number of pixels contained in a trace, dominates the processing time of the entire RO (Fig. 7).

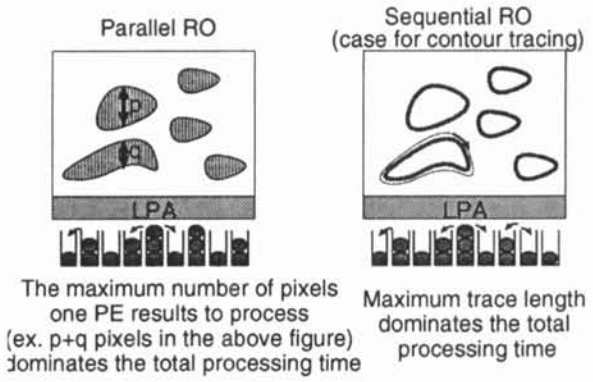


Figure 7: Performance aspects of the *stack-based* method for ROs

4 Overhead Estimation of Parallelizing Methods on LPAs

Each of the five basic parallelizing methods described in the previous section results to provide a pixel-updating-wave that sweeps through the entire source image or all regions within the source image (Fig. 8). However, the direction and speed of each pixel-updating-wave differs, as the control overhead for proceeding each wave forward an unit pixel distance is different between parallelizing methods. Control overhead is less for *row*, *column*, and *row-systolic* methods than for *slant-systolic* and *stack-based* methods. Generally control overhead for *stack-based* method is larger than that for *slant-systolic* method, as dynamic propagation and detection of region pixels performed by the former are usually a heavier task than static scheduling of the pixel processing order performed by the later.

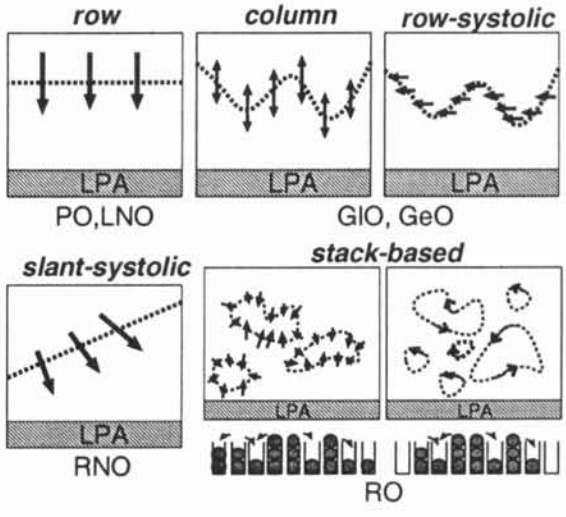


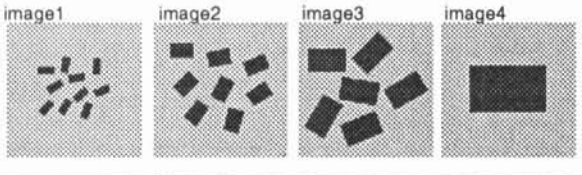
Figure 8: Pixel-updating-waves for each basic parallelizing method

By taking into account the control overhead described above, selection can be made between *stack-based* and *row* method for *parallel RO*, or between *stack-based* and *slant-systolic* method for *sequential RO*, according to the sizes of regions to be processed. The control overhead of the *stack-based* method is now trading off with the structural overhead, that is, the overhead for *row* or *slant-systolic* method to operate on unnecessary pixels (pixels not belonging

to any region), and to neglect the discovery of other *ready* pixels (pixels which have already fulfilled the imposed pixel updating order constraints).

Table 1 shows the processing times on IMAP-VISION, using respectively *stack-based* and *slant-systolic* method, to perform the previously described two-scan distance transform, a RNO and together a *sequential RO* if regarding groups of foreground pixels as regions. Programs are written in 1DC. The four 256×256 test images being used have a gradually increasing region sizes. The RO specific functions used for the *stack-based* implementation of the forward scan part of the distance transform are shown in 1DC in the following as an example.

```
#define Poped      0
#define Finished   1
#define Pushed     2
sep unsigned char Img[NROW],Tmp[NROW],Stack[NROW/2];

sep unsigned char IsSeed(int i)
{
   sep unsigned char r, IsContourPixel();

   mif (Img[i]) Tmp[i] = Poped;
   melse        Tmp[i] = Finished;

   mif (r=IsContourPixel(Img,i))  Tmp[i] = Pushed;
   return r;
}

/* use dt() as Pixel_op() */
#define Pixel_op(x,img)  dt(x,img)

void pnbh4(sep unsigned char x)
{
   sep unsigned char a,b,c,d,e;

   a= :>Tmp[:<x-1];b= Tmp[x-1];c= :<Tmp[:>x-1];
   d= :>Tmp[:<x];  e= Tmp[x];

   mif ((e==Poped) && (a & b & c & d)==Finished)){
        Stack[ss++]=x;
        Tmp[x] = Pushed;
   }
}

void Push_nbh_ptrs(sep unsigned char x)
{
                              pnbh4(:>x);
   pnbh4(:<x+1); pnbh4(x+1); pnbh4(:>x+1);
}
```



method	image1	image2	image3	image4
stack-based	4.8ms	6.1ms	7.9ms	10.1ms
slant-systolic	8.0ms			

Table 1: Processing time of two different parallelizing methods for the distance transform operation

According to Table 1, as the region sizes grow, the control overhead of the *stack-based* method gradually overcomes the structural overhead of the *slant-systolic* method. This result implies architectural subjects of LPAs for reducing control overheads produced by the parallelizing methods, especially those produced by the *stack-based* method. An even better performance for ROs on LPAs can be achieved if the subjects can be adequately solved in the near future.

5 Conclusion

In this paper, a data parallel language succinctly designed for a virtual LPA, and its compiler for an existing LPA, are first described. Then, a guideline for parallel algorithm development using the language, which consists of five basic parallelizing method: *row, column, row-systolic, slant-systolic*, and *stack-based*, are provided. Each category of low to intermediate level image operations has shown to be efficiently implemented on LPAs using each or a combination of the parallelizing methods. Furthermore, overhead produced by the two parallelizing methods, *slant-systolic* and *stack-based*, are discussed and compared. A conclusion is that, further improvement of performance on LPAs for region operations can be achieved, by architectural supports for reducing the control overhead of the *stack-based* method.

References

[1] T.J.Fountain, "The CLIP7A Image Processor," IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI), Vol.10, No.3, pp.310–319,1988.

[2] L.A. Schmitt et al.,"The AIS-5000 Parallel Processor," IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI), Vol.10,No.3, pp.320–330,1988.

[3] Y. Fujita et al.,"IMAP: Integrated Memory Array Processor," Journal of Circuits, Systems and Computers", Vol.2, No. 3, pp.227–245, 1992.

[4] Y. Fujita et al.,"IMAP-VISION: An SIMD Processor with High-Speed On-chip Memory and Large Capacity External Memory", Proc. of IAPR Workshop on Machine Vision Applications (MVA), Nov. 1996.

[5] P. P. Jonker, "Architectures for Multidimensional Low- and Intermidiate Level Image Processing," Proc. of IAPR Workshop on Machine Vision Applications (MVA), pp.307–316, 1990.

[6] G. Borgefors, "Distance Transformations in Digital Images," Computer Vision, Graphics, and Image Processing, Vol.34, pp.344–371, 1986.

[7] S.Kyo et al., "Efficient Implementation of Image Processing Algorithms on Linear Processor Arrays using the Data Parallel Language 1DC and its Compiler", Technical Report of Information Processing Association of Japan (IPSJ), Computer Architecture Section 119-17, pp.95–100, Aug. 1996. (In Japanese)

[8] M.Kass et. al,"Snakes: Active Contour Models", International Journal of Computer Vision,321-331 (1988).

[9] P.E Danielsson, "Parallelsim in Low Level Vision Algorithms and Architectures", Proc. of the 6th Scandinavian Conference on Image Analysis (SCIA), Finland, pp.25–31, 1989.

[10] A.Tanaka et.al, "A Rotation Method for Raster Image Using Skew Transformation", Proc. IEEE Conf. on Computer Vision and Pattern Recognition, Miami, pp.272–277, 1986.

[11] D.R.Helman et.al, "Efficient Image Processing Algorithms on the Scan Line Array Processor", IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI), Vol.17, pp.47–56, Jan. 1995.