A New Parallel Algorithm and Its Simulation on A Hypercube Simulator for Lowpass Digital Image Filtering Using Systolic Array

Received on April 7, 1997  Accepted for publication on May 9, 1998

Abdulraouf Al-Hallaq*  Saad Amin**

Abstract

This paper introduces a new parallel algorithm and its simulation on a hypercube simulator for the lowpass digital image filtering using a systolic array. This new algorithm is faster than the old one (Amin, 1988). This is due to the fact that the old algorithm carries out the addition operations in a sequential mode. But in our new design these addition operations are divided into two groups, which can be performed in parallel. One group will be performed on one half of the systolic array and the other on the second half, that is, by folding. This parallelism reduces the time required for the whole process by almost quarter the time of the old algorithm.

2. Introduction

Image processing is one of the areas in which most of the tasks are computationally intensive due to the large amount of data and the time-consuming repetitive operation of the algorithms in use. For example, a colored image of the size 1024*1024 requires 2M bytes,
assuming each color has 32 gray levels. The image goes through several levels of processing from the level of the numeric operations to the level of the symbolic operations. At each level the algorithm is time consuming. Therefore, the computational demand is far above the capacity of the conventional computers especially in the case of the real time applications where a sequence of images should be processed in a very short time. To speed up the operation, several parallel architectures and processing methods have been discussed at various levels of the image processing (Yalamanchili, et al., 1985). They may be classified into special-purpose (Duff, 1978, Batcher, 1980, Luetjen, 1980) or general-purpose (Narendra, 1981; Sternberg, 1981). One of the tasks in image processing that may be processed in parallel is the image smoothing operation using a low-pass digital image filtering (Bombardieni, 1992; Change & Lin, 1992). It is possible to implement such a parallel algorithm on a printed board or a chip due to the dramatic development of VLSI technology. Following kung's systolic concept (Kung, 1982; Li & W, wah 1985 Kung & Leiserson, 1979), many systolic arrays have been proposed to solve various problems (Walter, 1993; Boriokoff, 1994). These array processors generally consist of a regular array of simple and identical processing elements (PE's) which rhythmically compute and pass data through the system. The systolic system provides a realistic model of computation, which captures the concepts of pipelining, parallelism and interconnection structure. The purpose of this paper is to introduce a new parallel algorithm using a systolic design for the low-pass digital image filtering which can be implemented as an Occam program on a Transputer Development System, and to compare it with the old design proposed in Amin, 1988). And because such transputer is not available we simulate the algorithm and run it on the hypercube simulator (see section 6).
3- Literature Review:

3.1- The 2-D Convolution: The two dimension convolution is represented by the following equation where h has non zero for \(-w < k < w\) and \(-v < l < v\) and zero else where.

\[
y(i, j) = \sum_{k=-w}^{w} \sum_{j=-v}^{v} f(k, i)h(i - k, j - l)\]

Equation (1) says that the output \(y(i,j)\) at the point \((i,j)\) is given by the weighted sum of the input pixels that surrounds \(i\), where the weights are given by \(h(k,l)\), which is generated by a series of shift, multiply and sum operations. The values of \(h\) are also referred to as the filter weight, the filter kernel, or the filter mask.

3.2- Parallel Algorithm For Low - Pass Digital Image Filtering And The 2-D Convolution:

From a computational point of view, an efficient way of computing a 2-D convolution is to transform it into that of computing a 1-D convolution. Therefore, the 2-D convolution can be performed on a systolic array for a 1-D convolution.

Consider an image matrix of size \(n\times n\) and a kernel \(w\) of size \(m\times m\), Figure (1) shows the image and the kernel for the case \(n=5\) and \(m=3\), then each row \(i\) of the image can be represented as:

\[
x_{i1},
\]

where \(x_{i1} = x_{i1}, x_{i2}, ... , x_{in}\), \(i=1..n\)

and the image is represented as:

\[
x = x_{11}, x_{21}, ... , x_{n1}
\]

with a total length of \(n^2\).
The 2-D convolution defined in equ. (1) can be viewed as a 1-D convolution as follows:

\[
\begin{array}{cccccc}
  x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\
  x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \\
  x_{31} & x_{32} & w_{11} & w_{12} & w_{13} \\
  x_{33} & x_{34} & w_{21} & w_{22} & w_{23} \\
  x_{41} & x_{42} & w_{31} & w_{32} & w_{33} \\
  x_{51} & x_{52} & x_{53} & x_{54} & x_{55}
\end{array}
\]

(a) 2-D convolution

\[
x = x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{31}, ..., x_{54}, x_{55}
\]

\[
w = w_{11}, w_{12}, w_{13}, 0, 0, w_{21}, w_{22}, w_{23}, 0, 0, w_{31}, w_{32}, w_{33}
\]

(b) 1-D convolution

Figure (1) a) 2-D convolution, b) 1-D convolution, n=5, m=3.

Following the same procedure, each row of the kernel can be represented as:

\[
w_{i*} = w_{i1}, w_{i2}, ..., w_{im}, \quad i = 1 .. m
\]

and the kernel as:

\[
w = w_{1*}, (n-m)!*[0], w_{2*}, (n-m)!*[0], ..., w_{m*}
\]

where \((n-m)! [0]\) means a repetition of \(n-m\) zeros inserted to make each row of the kernel to be of the same length as the length of each
Parallel Processing

row of the image matrix. Thus, the total length of the 2-D kernel when transformed into a vector is $n(m-1)+m$.

At this point, we can relax the constraint stipulating that the input sequence is formed by the rows of the input array entered one after another into the systolic array without any delay between its consecutive rows.

Now, following Kung's method but rather than viewing the kernel as sliding over the image, we consider the kernel as stored in reverse order in the systolic array of $n(m-1)+m$ components one at every component from the right to the left, that is as:

$$w_{33}, w_{32}, \ldots$$

We also consider that the image is sliding from left to right at every pulse of the network. The y's are sliding at the same speed as the x's of the image but in the opposite direction, from right to left, and the $n(m-1)+m$ multiplications are performed simultaneously and stored in y.

Also, each processor has an input register to receive the x's, a register to hold the kernel value, a multiplier and an adder to receive the accumulated y's that result from its right neighbor and to add it to the result of its multiplication, then passes it to its left neighbor, see Figure (2). When the result accumulates its final value $y_s$, it will be outputted to host-1 at the left where

$$y_s = \sum_{i,j=1}^{m} W_{i,j} X_{i+ls/(n-m+1)l+j+smod(n=m+1)}$$
4- The New Parallel Algorithm For Low-pass Digital Image Filtering:

4.1- Problems in the Old Algorithm:

It can be easily seen that the old algorithm has a major problem when it reaches to the addition operations. The problem arises because of the addition operations are done at every component one after another, so no parallelism is performed in these operations. Following the same concept of computing the 2-D convolution by transforming it into a 1-D convolution and keeping the kernel stationary as before, it turns out that if the y's slide in either direction, then there is no gain in speed. This is due to the fact that after each of the \(n(m-1)+m\) x's arrives at the corresponding component and all the \(n(m-1)+m\) multiplications are carried out simultaneously, the addition operations will be carried out in a pipelining manner one after another. In other words, the addition operations will be performed sequentially.

4.2- Parallelism in the New Algorithm by Folding:

The new algorithm is based on the same concept as the old one except that rather than letting the y's slide only from the opposite direction with respect to the x's, we let the right half of the x's to return
from the most right component to the left after being multiplied. This is because of the left half of the PE's are sitting idle. In other words, to speed up the addition operations and hence the whole process and to make use of those PE's which are sitting idle, we divide the \( n(m-1)+m \) components into two groups and let the most right component sends its result of multiplication to its left neighbor and the most left component sends its result to its right neighbor. During this operation, the additions are accumulated and then moved in both directions (we call this operation folding). This makes the additions to be performed in parallel in almost half the time required by the old design. When the two added final results of the two halves reach the extra middle component, one of them is delayed until the other is entered then the first one will be entered. This will require an additional time unit. After that they will be added up and the result of the addition is produced to the host after \( \frac{1}{2}(n(m-1)+m)+1 \) time units which is almost half the \( n(m-1)+m \) time units taken by the old method at the expense of an extra component, the middle component. See Figure (3).

In Figure (3), the x's slide left to right. The y's slide in two directions, \( y_1...y_6 \) slide from right to left and \( y_7...y_{12} \) and \( y_{13} \) slide from left to right. As seen in Figure (3) \( y_1 \) gets the result of multiplying \( x_{11} \) and \( w_{11} \) at the right.

\[
\begin{align*}
&w_{33} \quad w_{32} \quad w_{31} \quad 0 \quad 0 \quad w_{23} \quad w_{22} \quad w_{21} \quad 0 \quad 0 \quad w_{13} \quad w_{12} \quad w_{11} \\
x_{11} \\
x_{12} \quad x_{11} \\
x_{13} \quad x_{12} \quad x_{11} \\
\vdots \quad \vdots \quad x_{11} \\
\vdots \quad \vdots \quad \vdots \\
\vdots \quad \vdots \\
x_{33} \quad x_{32} \quad x_{31} \quad x_{25} \quad x_{24} \quad \ldots \ldots \quad x_{15} \quad x_{14} \quad x_{13} \quad x_{12} \quad x_{11}
\end{align*}
\]
most component and goes to the second component from the right in which the value of \( x_{12} \) is added to it. The same is done at the left side of the network where \( y_7 \) gets the result of multiplying \( x_{33} \) and \( w_{33} \) at the left most component and goes to the second component from the left. When both of the accumulated results of \( y_6 \) and \( y_{12} \) come out of the two components surrounding the middle component they enter the extra (the one above the network in Figure 3b) component to be added and the result is then outputted. We would like to mention here that the \( x \)'s are shifted off the network at the right and that the \( x_{11} \) in the second array from the right does not exist in reality but is shown for clarity. This value has already been multiplied and stored in \( y_1 \).
5- Performance and Comparison:

Consider the matrix of size nxn and the kernel of size mxm as above. Now, assume any operation takes one unit of time. Then in the old and the new algorithm, the elements of the matrix that are needed to be convoluted for the first time need nm-(n-m) time units to reach their corresponding components. Also a one time unit for the multiplication is needed for the old and the new algorithms. Now, the addition and shifting will take another n(m-1)+m units in the old algorithm. Thus, the total number of time units needed in the old design is 2[n(m-1)+m]+1. But in our new algorithm, the addition and the shifting need only 1/2[n(m-1)+m]+1 units of time. This is because the two groups to be added are added at the same time in parallel, one group is added by a one half of the systolic array and the other group is added by the other half of the array. The extra time unit corresponds to the addition operation at the extra component of the systolic array. Thus, the total number of units needed in our new algorithm is only 3/2[n(m-1)+m]+2. So, the total time for the whole process in the new algorithm is reduced to almost 3/4th the time taken by the old algorithm.

6- Hypercube Simulation:

This new algorithm is implemented and run on the Hypercube simulator. It gives a speed up in the addition and shifting operations that are done in parallel according to what we suggested in section 5. But because the simulator gives an approximated time, the speed up operation is calculated as follows:

Suppose the addition and shifting in each processor takes 1 msec, and the number of the processors is r. The time needed to perform the addition and shifting operations will be 1/2r+1. Program 1 shows the host and program 2 shows the nodes program.
Program-1: The Host Program:
#define HOSTPID 100
#define ALL_node -1
#define NPID 0
#define msg_14 10
#define msg_host -1
int om[4][4];
int msg;
int res;
int nodeno;
long no_nodes,
v1,v2;
int i,j;
unsigned long ti;
main()
{
for (i=0;i<=3;i++)
for (j=0;j<=3;j++)
om[i][j]=1;
setpid(HOSTPID);
load("node",ALL_node,NPID);
no_nodes=numnodes();
/* ti=mclock(); */
nodeno = 0;
for (i=0;i<=3;i++)
for (j=0;j<=3;j++)
{
msg=om[i][j];
printf("%d to nodeno host.c %d \n",msg,nodeno);
csend(msg_14,&msg,sizeof(msg),nodeno,NPID);
nodedo = nodedo+1;
}
crecv(msg_host,&v1,sizeof(v1));
printf("%d from nodehost %d\n",v1,infonode());
crecv(msg_host,&v2,sizeof(v2));
printf("%d from nodehost1 %d\n",v2,infonode());
/*ti=mclock()-ti;*/
printf("%d \n",v1+v2);
}

Program-2: The Node Program:
#define msg_14 10
#define msg_node 10
#define host_pid 100
#define msg_host -1
    int msg;
    int res;
    int mres;
long my_node,v1,v2,
    next_node,
    my_pid,
    no_host,
    next_pid;
int a,b,res1;
int co=0;
main()
{
    my_node=mynode();
    my_pid=mypid();
    next_pid =my_pid;
b=5;
if (my_node == 0) next_node=1;
if (my_node == 1) next_node=3;
if (my_node == 2) next_node=6;
if (my_node == 3) next_node=2;
if (my_node == 4) next_node=5;
if (my_node == 5) next_node=7;
if (my_node == 6) next_node=4;
if (my_node == 7) next_node=myhost();
if (my_node == 8) next_node=myhost();
if (my_node == 9) next_node=11;
if (my_node == 10) next_node=8;
if (my_node == 11) next_node=10;
if (my_node == 12) next_node=13;
if (my_node == 13) next_node=9;
if (my_node == 14) next_node=12;
if (my_node == 15) next_node=14;
    crecv(msg_node,&mres,sizeof(mres));
if (my_node == 15)
{
    mres = mres + msg*b;
    printf("%d from 15 node %d \n", mres ,my_node);
    csend(msg_node,&mres,sizeof(mres),next_node,next_pid);
}
if (my_node == 0)
{
    mres = mres + msg*b;
    printf("%d from 0 node %d \n", mres ,my_node);
    csend(msg_node,&mres,sizeof(mres),next_node,next_pid);
}
if (my_node >0 & & my_node <8)
{ 
mres=mres+msg*b;
crecv(msg_node,&mres,sizeof(mres));
printf("%d from >0 <8 node %d \n",mres,my_node);
if(my_node < 7)
{
csend(msg_node,&mres,sizeof(mres),next_node,next_pid);
printf("%d my_node %d next \n",my_node,next_node);
}
else
{
    v1=mres;
    no_host=myhost();
csend(msg_host,&v1,sizeof(v1),no_host,host_pid);
}
}
if (my_node > 8 && my_node <15)
{
crecv(msg_node,&mres,sizeof(mres));
mres=mres+msg*b;
printf("%d from node %d \n",mres,my_node);
if(my_node > 8)
csend(msg_node,&mres,sizeof(mres),next_node,next_pid);
else
{
    v2=mres;
    no_host=myhost();
csend(msg_host,&v2,sizeof(v2),no_host,host_pid);
}
}
}
7. Conclusion:

A new parallel algorithm using systolic array for a low-pass digital image filtering is presented. This new algorithm is faster and takes 3/4th the time of the old algorithm at the expense of an extra component of the systolic array, assuming one unit of time for any operation. This can be implemented and tested by an Occam program that runs on a Transputer Development System (TDS). This work may be applied for the Laplacian and other high-pass filters.

References


خوارزمية متوازية جديدة ومحاكاتها على حاسوب هايبركوب
شرح الترددات المنخفضة للصور الرقمية باستخدام
المصفوفة الإنتقابية

عبدالرؤوف الحلاق، الجامعة الأردنية، عمان - الأردن
سعد أمين، جامعة لبيرو، بريطانيا

ملخص
يقدم هذا البحث خوارزمية متوازية جديدة لرشح الترددات المنخفضة للصور الرقمية
иمحاكاتها على حاسوب هايبركوب باستخدام المصفوفة الإنتقابية. هذه الخوارزمية
الجديدة أسرع من الخوارزمية القديمة الواردة في المرجع 14. وتعزى هذه السرعة إلى أن الخوارزمية
القديمة تجري عمليات الجمع بطريقة التتابعية. ولكن في هذه الخوارزمية الجديدة فإن عمليات الجمع
قسمت إلى مجموعتين ليتم جمع المجموعتين على التوازي، ويتم جمع احدي المجموعتين بتصف
المصفوفة الإنتقابية والجموعة الأخرى بالنصف الآخر، أي بطريقة الطي. ويعمل هذا التوازي على
اختصار الزمن اللازم لإجراء كل مرحلة عملية الترشيح بما يقارب ربع الزمن الذي تستغرقه
الخوارزمية القديمة.