IMAGE TEMPLATE MATCHING ON MIMD HYPERCUBE MULTICOMPUTERS[+]

Sanjay Ranka[*] and Sartaj Sahni

University of Minnesota

Abstract

Efficient algorithms for image template matching on fine grained as well as medium grained MIMD hypercube multicomputers are developed. The medium grained MIMD algorithm is developed specifically for the NCUBE multicomputer. This algorithm is compared experimentally with an algorithm that is optimized for the Cray 2 supercomputer.

Keywords and Phrases

Hypercube multicomputer, image template matching, Kirsch templates, MIMD multicomputers, one and two dimensional convolution

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374
[*] Professor Ranka's current address is School of CIS, 4-192 Center for Science & Technology, Syracuse University, Syracuse, NY 13244-4100.

1 INTRODUCTION

The inputs to the image template matching problem are an $N \times N$ image matrix $I[0..N-1, 0..N-1]$ and an $M \times M$ template $T[0..M-1, 0..M-1]$. The output is an $N \times N$ matrix C2D where

$$C2D[i, j] = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} I[(i+u) \bmod N, (j+v) \bmod N] * T[u,v], \quad 0 \le i, j < N$$

$C2D$ is called the two dimensional convolution of I and T. Template matching, i.e., computing $C2D$, is a fundamental operation in computer vision and image processing. It is often used for edge and object detection; filtering; and image registration [ROSE82, BALL85]. Throughout this paper we assume that N and M are powers of 2.

Because of the fundamental nature of this problem and because of its high complexity ($O(M^2N^2)$ on a single processor computer), much attention has been devoted to the development of efficient fine grain multicomputer parallel algorithms. For example, Chang, Ibarra, Pong and Sohn [CHAN87] have studied this problem on an SIMD pyramid computer; Ranka and Sahni [RANK88a], Maresca and Li [MARE86], and Lee and Agarwal [LEE87] have considered mesh connected computers; and Fang, Li and Ni [FANG85], Fang and Ni [FANG86], Prasanna Kumar and Krishnan [PRAS86] and Ranka and Sahni [RANK88b] have considered SIMD hypercube multicomputers. In this paper, we restrict our attention to MIMD hypercube multicomputers. We consider both fine and medium grained MIMD hypercubes.

The algorithms developed in [PRAS87], [LEE87], [FANG85], [FANG86], and [MARE86] for fine grained SIMD hypercubes make significant use of the O(1) data broadcast capability available from host to hypercube processors in this model. This permits the transfer of one unit of data from the host to all processors of the hypercube in O(1) time. This broadcast capability is not available in the MIMD model. However, in the MIMD model it is possible for different processors to simultaneously route along different dimensions of the hypercube. Using this feature we develop algorithms for the two cases considered in [PRAS87]. Both of these cases assume N^2 processors are available. However, in one case each processor has O(M) memory while in the other each has O(1) memory. The asymptotic complexity of our algorithms for both cases is

O(M^2 + log$N$). The algorithms of [PRAS87] take O(M^2 + log$N$) and O(M^2log*M + log$N$) time for the two cases, respectively. Our algorithms are simpler and have smaller constant factors than those of [PRAS87]. It should be noted that an O(M^2 + log$N$) time algorithm for the case of O(1) memory per processors has been developed in [RANK88b]. Our algorithm for this case on an MIMD model is much simpler than the algorithm deveoped in [RANK88b] and is faster by a constant factor. Since the only MIMD feature used by us is the ability of different processors to communicate along different dimensions of the hypercube in the same time step, we can run our algorithms on an SIMD model that has been extended to support this capability.

Using the techniques of [PRAS87], both our algorithms may be generalized to obtain asymptotically optimal algorithms for MIMD hypercube computers with N^2K^2, $1 \le K \le M$ processors and $O(M/K)$ and $O(1)$ memory per processor respectively.

Our medium grain MIMD algorithm is developed for the NCUBE hypercube. This algorithm is evaluated experimentally and a comparison with a single processor Cray 2 is made. The time taken by a 64 processor NCUBE hypercube is about five times that required by a single processor Cray 2 supercomputer.

Section 2 describes our hypercube model. In addition, notation and some fundamental data movement operations are described in this section. In Section 3, we develop fine grained algorithms for one dimensional convolution. These form a basic component of our two dimensional convolution algorithms which are developed in Section 4. The medium grain MIMD algorithm for the NCUBE is described in Section 5. Experimental results are also presented in this section.

2 PRELIMINARIES

2.1 Hypercube Multicomputer

The important features of an MIMD hypercube and the programming notation we use are:

1. There are $P = 2^p$ processing elements connected together via a hypercube interconnection network. Each PE has a unique index in the range $[0, 2^p - 1]$. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}....i_0$ be the binary representation of the PE index i. Let

$\overline{i_k}$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit. I.e., processor $i_{p-1}i_{p-2}...i_0$ is connected to processors $i_{p-1} \cdots \overline{i_k}....i_{0,}$ $0 \le k \le p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

2. The local memory of each PE holds both the data and the program that the PE is to execute. Throughout this paper, we shall uses brackets([ ]) to index an array and parentheses ('( )') to index the PEs. Thus A[*i*] refers to *i*'th element of the array A while A(*i*) refers to the A register of PE *i*. Likewise $A[i](j)$ refers to the *i*'th element of array A of PE *j*.

3. At any given instance, different PEs may execute different instructions. In particular, PE *i* may transfer data to PE $i^{(b)}$, while PE *j* simultaneously transfers data to PE $j^{(a)}$, $a \neq b$.

4. An *instruction mask* is a boolean function used to describebPEsb which will remain active duringban instruction. For example, in the instruction

   $A(i) := A(i) + 1, \quad (i_0 = 1)$

   $(i_0 = 1)$ is a mask,bwhich states that only PEs with index bit 0 equal to 1 remain active during the instruction. I.e., odd indexed PEs increment their A register value by 1. We shall often omit the PE index from our instructions. Thus, the above statement can also be written as

   $A := A + 1, \quad (i_0 = 1)$

5. In a *unit route*, data may be transmitted from one processor to another to which it is directly connected. We assume that the links in the interconnection network are unidirectional. Hence at any given time, data can be transferred either from PE *i* $(i_b = 0)$ to PE $i^{(b)}$ or from PE *i* $(i_b = 1)$ to PE $i^{(b)}$. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

2.2 Hypercube Embedding of a Grid

Figure 1 shows an embedding of a 4×4 image grid into a hypercube of dimension 4. The number inside a box is the binary representation of the index of the PE to which that element is mapped. This embedding uses the binary reflected gray code mapping of [CHAN86]. An i bit binary gray code S_i is defined recursively as below:

$S_1 = 0, 1; \quad S_k = 0[S_{k-1}], 1[S_{k-1}]^R$

where $[S_{k-1}]^R$ is the reverse of the $k-1$ bit code S_{k-1} and $b[S]$ is obtained from S by prefixing b to each entry of S. So, S_2 = 00, 01, 11, 10 and S_3 = 000, 001, 011, 010, 110, 111, 101, 100.

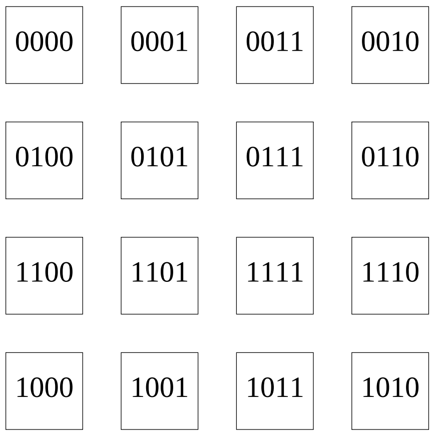

Figure 1: Embedding of a 4×4 mesh in a hypercube of dimension 4

If $N = 2^n$, then S_{2n} is used to map an N $\times$ N grid into a $P = N^2$ hypercube. The elements of S_{2n} are assigned to the elements of the $N \times N$ grid in a snake like row major order [THOM77]. This mapping has the property that grid elements that are neighbors are assigned to neighboring hypercube nodes. Another interesting property is evident from the definition of S_k and the linear drawing of Figure 2. In this figure, PEs appear in the order given by S_k. A hypercube has circular lists of length 2^i for all i embedded in it. Furthermore, these circular lists of length 2^i are present in every row and column of the grid embedding. Also, the PEs in each circular list of length 2^i form a 2^i PE subhypercube, $i > 1$.

For a hypercube with $P = 2^p$ PEs, we define the function $gray(i)$ such that $gray(0) = 0$ and $gray(i)$ is the index of the PE that immediately follows the PE $gray(i-1)$ in the circular list of size 2^p obtained from the above gray code embedding. For the example of Figure 2, $P = 2^3 = 8$, $gray(0) \cdots gray(7)$ = (0, 1, 3, 2, 6, 7, 5, 4). The function *igray* is the inverse of *gray*. So,

$igray(0) \cdots igray(7) = (0, 1, 3, 2, 7, 6, 4, 5)$.

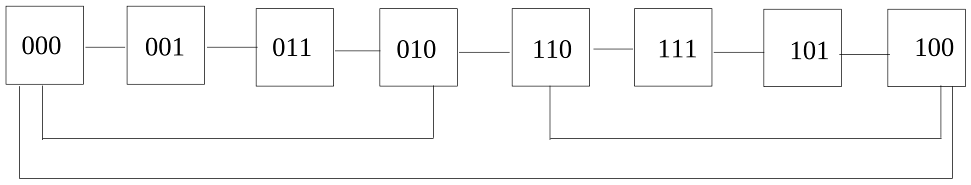

Figure 2: Rings of size 2, 4 and 8 in an 8 PE hypercube

2.3 Fundamental Operations

2.3.1 Data Sum

Assume that W is a power of 2 and that a P processor hypercube is tiled by windows of this size such that each window forms a subhypercube with W PEs. The data in each of the windows is to be summed and the sum left in a prespecified PE (same relative PE for each window). For example, if we are summing the A register data, we may be required to compute:

$$Sum(index(iW)) = \sum_{j=0}^{W-1} A(index(iW + j)), \quad 0 \le i < (P/W)$$

Here, $index(q)$ gives the physical index of the q'th PE in the tiling scheme. We assume that the P PEs are first ordered (for example using an S_r) and then tiled using $1 \times$ W tiles. Thus, $iW + j$ is the j'th PE in the i'th tile and iW is the 0'th PE in the i'th tile. Data sum can be done in log W unit routes [DEKE81].

2.3.2 Shift

$SHIFT(A,i,W)$ shifts the A register data circularly counter-clockwise by i in windows of size W. I.e., $A(gray(qW+j))$ is replaced by $A(gray(qW + (j-i) \bmod W))$, $0 \le q < (P/W)$, $0 \le j < W$. In a gray code indexing, the indexing within each size 2^j window also corresponds to a gray code (consider the least significant j bits). Hence each pair of adjacent size 2^j windows differs in exactly one bit. Now suppose the shift amount i is a power of 2. We can get data to the correct size i window by routing along the single bit in which adjacent size i windows differ. Following this, the data in each size i window needs to be reversed (unless $i = 1$). This reversal may be accomplished by exchanging data in the two size $i/2$ windows that make up a size i window. The total number of unit routes required when i is a power of 2 is therefore at most 2 to get the data to the correct size i window (note that 2 routes are needed when $i = W/2$ and one otherwise) plus at most 2 to reverse within the size i window. Hence at most 4 unit routes are needed to perform a shift of size i. When i is not a power of 2, i can be written as the sum of powers of 2 and the shift obtained by performing successive power of 2 shifts. Since only one of these can be a $W/2$ shift, the number of unit routes is at most $3\#1(i) + 1$, where $\#1(i)$ is the number of ones in the binary representation of i. The worst case performance can be kept at $3(logW)/2 + 1$ by noting that if there are more than $(logW)/2$ one bits, we can do a $W-1-i$ clockwise shift followed by a unit clockwise shift. Also note that the special cases of $i = 1$, 2, and 3 are easily done in i unit routes unless $W = 1$ (in this case, a shift of 1 takes 2 unit routes).

2.3.3 Data Accumulation

For this operation, PE j has an array $A[0..M-1]$ of size M. In addition, each PE has a value in its I register. After the data accumulation, the M elements of A in each PE j are such that:

$$A[i](gray(j)) = I(gray((j+i) \ mod \ P)),\ 0 \le i < M,\ 0 \le j < P$$

This can be accomplished in M-1 unit routes (for P > 2) by repeatedly shifting by -1 in windows of size P. Let $ACCUM(A, I, M)$ be the procedure that does this.

2.3.4 Adjacent Sum

This operation is defined in [PRAS87]. For each PE, p, $0 \le p < P$, the sum

$$T(gray(p)) = \sum_{i=0}^{M-1} A[i](gray(p+i) \ mod \ M))$$

is to be computed.

As mentioned earlier, every hypercube of size P can be viewed as consisting of P/M subhypercubes (blocks) each of size M. For every PE p, some (or all) of the A’s needed to compute

$T(gray(p))$ are in the block containing PE p. The remainder are in the next block of PEs. The strategy to compute T is as follows:

1) Each PE, p, begins with two variables S and T (initially 0). These values circulate through the M PEs in the block. T accumulates the A values in the block needed in the sum for $T(gray(p))$. S accumulates the A values needed for $T(gray((p-M) \ mod \ P))$.

2) The S values are shifted clockwise by M positions and added to the T values.

The formal algorithm is given in Figure 3. The number of unit routes is $2M + 4$ (recall that M is a power of 2 and a power 2 shift takes at most 4 unit routes). This can be reduced to $M + 4$ by shifting S and T as a single packet.

```
procedure AdjacentSum(A, M)
begin
  S:= 0; T:=0;
  for i := 0 to M-1 do
  begin
    T(p) := T(p) + A[i](p); ($igray(p)$ mod M ≥ i)
    S(p): = S(p) + A[i](p); ($igray(p)$ mod M < i)
    SHIFT(T, 1, M);
    SHIFT(S, 1, M);
  end
  SHIFT(S, -M, P);
  T := T + S;
end {of AdjacentSum}
```

Figure 3: Adjacent Sum

3 ONE DIMENSIONAL CONVOLUTION

The inputs to the one dimensional convolution problem are vectors $I[0..N-1]$ and $T[0..M-1]$. The output is the vector C1D where:

$$C1D[i] = \sum_{v=0}^{M-1} I[(i+v) \ mod \ N]*T[v] \quad , 0 \le i < N$$

We use the computation of C1D as a basic step in our algorithms to compute C2D. In this section, we develop algorithms for C1D. Our algorithms assume that there are $P = N$ processors and that the vector I is mapped onto the hypercube using the gray code mapping (i.e., $I[i]$ on PE $gray(i)$) . Further, we assume that there are N/M copies of T in the hypercube with one copy in each block of M processors. Within a block, the mapping of T is the same as that of I.

```
procedure C1D_M
{O(M) memory algorithm for one dimensional convolution}
begin
    ACCUM(A, I, M);
    b := igray(p) mod M; {relative index of PE in M block}
    C1D := 0;
    for j := 1 to M do
    begin
        C1D := C1D + A[b] * T;
        b := (b+1) mod M;
        SHIFT(T, -1, M);
    end
end; {of C1D_M}
```

Figure 4: $O(M)$ memory computation of C1D

3.1 O(M) Memory

When each processor has $O(M)$ memory, the most effective way to compute C1D is to first perform a data accumulation on I. Following this, each processor has all the I values needed to compute the corresponding entry of C1D. Next, the T values are circulated through each block of M processors. During this circulation, the T values are multiplied by the I values and the C1D values computed. Procedure C1D_M (Figure 4) provides the details. Note that while the final shift on T is not necessary for the computation of C1D, our algorithms for C2D assume that T is unchanged by the C1D algorithms. This final shift restores the original T values. The number of unit routes is 2M.

3.2 O(1) Memory

When only O(1) memory per PE is available, we begin by first pairing I values in the processors. The pair in processor p is (A(p), B(p)) = (I $[(jM + 2k) \, mod \, N]$, I$[(jM + 2k + 1) \, mod \, N]$) where $i = igray(p)$, $j = \lfloor i/M \rfloor$, and $k = i \, mod \, M$. Figure 5 gives the initial AB pairs in each PE for the case $N = 16$, $M = 4$. This pairing operation is performed by first performing a related pairing operation in each block of M PEs. This related pairing operation obtains the configuration of the last column of Figure 6 and proceeds according to the pattern given in this figure. Figure 7

p	$i = igray(p)$	j	k	I	AB
0	0	0	0	I_0	$I_0 I_1$
1	1	0	1	I_1	$I_2 I_3$
3	2	0	2	I_2	$I_4 I_5$
2	3	0	3	I_3	$I_6 I_7$
6	4	1	0	I_4	$I_4 I_5$
7	5	1	1	I_5	$I_6 I_7$
5	6	1	2	I_6	$I_8 I_9$
4	7	1	3	I_7	$I_{10} I_{11}$
12	8	2	0	I_8	$I_8 I_9$
13	9	2	1	I_9	$I_{10} I_{11}$
15	10	2	2	I_{10}	$I_{12} I_{13}$
14	11	2	3	I_{11}	$I_{14} I_{15}$
10	12	3	0	I_{12}	$I_{12} I_{13}$
11	13	3	1	I_{13}	$I_{14} I_{15}$
9	14	3	2	I_{14}	$I_0 I_1$
8	15	3	3	I_{15}	$I_2 I_3$

$I_q = I[q]$

Figure 5: Initial AB pairs for N = 16, M = 4

shows the pattern for the case N= 16 and M = 8. If the related pairing is followed by a shift of -M/2 on the AB registers such that only PEs with the '-' values in the last columns of Figures 6 and 7 update their AB values, the desired pairing of I values is obtained. The formal algorithm is given in Figure 8.

The number of unit routes is at most 8 log M. This can be reduced to 4 logM by routing (A, B) pairs as single packets.

Once the AB pairing has been done C1D may be computed by rotating the AB values clockwise in a window of size P (in a single rotation, B's move to A's in the same PE and A's move to B's of the next PE) and rotating the T values clockwise in a window of size M. Figure 9 shows the initial AB pairs and T values for the case N = 16 and M = 4. Throughout the algorithm, the product of A(p) and T(p) will give one of the terms needed to compute C1D(*igray*(p)) for every PE p. B(p) will be the next I value needed. Initially, this is true for all processes except those with $igray(p) \; mod \; M = M - 1$. This situation is remedied by replacing B with I in these processors to get the first row labeled AB'. Following a rotation of AB, we get the second row labeled AB. Now, the B value in processors with $igray(p) \; mod \; M = M - 2$ needs to be changed to

$i = igray(p)$	M block	initial I	shift to B	shift -1	shift -1
				$i_0 = 1$	$i_1 = 1$
			B	AB	AB
0	0	I_0	I_0	$I_0 I_1$	$I_0 I_1$
1	0	I_1	I_1	-	$I_2 I_3$
2	0	I_2	I_2	$I_2 I_3$	-
3	0	I_3	I_3	-	-
4	1	I_4	I_4	$I_4 I_5$	$I_4 I_5$
5	1	I_5	I_5	-	$I_6 I_7$
6	1	I_6	I_6	$I_6 I_7$	-
7	1	I_7	I_7	-	-
8	2	I_8	I_8	$I_8 I_9$	$I_8 I_9$
9	2	I_9	I_9	-	$I_{10} I_{11}$
10	2	I_{10}	I_{10}	$I_{10} I_{11}$	-
11	2	I_{11}	I_{11}	-	-
12	3	I_{12}	I_{12}	$I_{12} I_{13}$	$I_{12} I_{13}$
13	3	I_{13}	I_{13}	-	$I_{14} I_{15}$
14	3	I_{14}	I_{14}	$I_{14} I_{15}$	-
15	3	I_{15}	I_{15}	-	-

$I_q = I[q]$

Figure 6: Related pairing for N = 16, M = 4

$i = igray(p)$	M block	initial I	shift to B	shift -1	shift -1	shift -2
				$i_0 = 1$	$i_1 = 1$	$i_2 = 1$
			B	AB	AB	AB
0	0	I_0	I_0	$I_0 I_1$	$I_0 I_1$	$I_0 I_1$
1	0	I_1	I_1	-	$I_2 I_3$	$I_2 I_3$
2	0	I_2	I_2	$I_2 I_3$	-	$I_4 I_5$
3	0	I_3	I_3	-	-	$I_6 I_7$
4	0	I_4	I_4	$I_4 I_5$	$I_4 I_5$	-
5	0	I_5	I_5	-	$I_6 I_7$	-
6	0	I_6	I_6	$I_6 I_7$	-	-
7	0	I_7	I_7	-	-	-
8	1	I_8	I_8	$I_8 I_9$	$I_8 I_9$	$I_8 I_9$
9	1	I_9	I_9	-	$I_{10} I_{11}$	$I_{10} I_{11}$
10	1	I_{10}	I_{10}	$I_{10} I_{11}$	-	$I_{12} I_{13}$
11	1	I_{11}	I_{11}	-	-	$I_{14} I_{15}$
12	1	I_{12}	I_{12}	$I_{12} I_{13}$	$I_{12} I_{13}$	-
13	1	I_{13}	I_{13}	-	$I_{14} I_{15}$	-
14	1	I_{14}	I_{14}	$I_{14} I_{15}$	-	-
15	1	I_{15}	I_{15}	-	-	-

Figure 7: Related pairing for N = 16, M = 8

```
procedure PAIRING(M)
{pairing I values in AB registers}
begin
   i := igray(p); {p is processor index}
   B := I;
   SHIFT(B, -1, P);
   A := I;
   for j := 1 to logM-1 do
   begin
      C := B; SHIFT(B, -$2^{j-1}$, M);
      B := C; ($i_j$ = 0)
      C := A; SHIFT(A, -$2^{j-1}$, M);
      A := C; ($i_j$ = 0)
   end
   C := B; SHIFT(B, -(M/2), P);
   B := C; ($i_{logM-1}$ = 0)
   C := A; SHIFT(A, -(M/2), P);
   A := C; ($i_{logM-1}$ = 0)
end; {of PAIRING}
```

Figure 8: Pairing of the I's

$I(p)$. With this insight, one arrives at the algorithm of Figure 10. Its correctness is easily established. The number of unit routes (including those for pairing) is at most $M + 8logM$.

i	I	AB	T	*AB´*	AB	T	*AB´*	AB	T	*AB´*	AB	T	*AB´*
0	I_0	I_0I_1	T_0	I_0I_1	I_1I_2	T_1	I_1I_2	I_2I_3	T_2	I_2I_3	I_3I_4	T_3	I_3I_0
1	I_1	I_2I_3	T_1	I_2I_3	I_3I_4	T_2	I_3I_4	I_4I_5	T_3	I_4I_1	I_1I_2	T_0	I_1I_2
2	I_2	I_4I_5	T_2	I_4I_5	I_5I_6	T_3	I_5I_2	I_2I_3	T_0	I_2I_3	I_3I_4	T_1	I_3I_4
3	I_3	I_6I_7	T_3	I_6I_3	I_3I_4	T_0	I_3I_4	I_4I_5	T_1	I_4I_5	I_5I_6	T_2	I_5I_6
4	I_4	I_4I_5	T_0	I_4I_5	I_5I_6	T_1	I_5I_6	I_6I_7	T_2	I_6I_7	I_7I_8	T_3	I_7I_4
5	I_5	I_6I_7	T_1	I_6I_7	I_7I_8	T_2	I_7I_8	I_8I_9	T_3	I_8I_5	I_5I_6	T_0	I_5I_6
6	I_6	I_8I_9	T_2	I_8I_9	I_9I_{10}	T_3	I_9I_6	I_6I_7	T_0	I_6I_7	I_7I_8	T_1	I_7I_8
7	I_7	$I_{10}I_{11}$	T_3	$I_{10}I_7$	I_7I_8	T_0	I_7I_8	I_8I_9	T_1	I_8I_9	I_9I_{10}	T_2	I_9I_{10}
8	I_8	I_8I_9	T_0	I_8I_9	I_9I_{10}	T_1	I_9I_{10}	$I_{10}I_{11}$	T_2	$I_{10}I_{11}$	$I_{11}I_{12}$	T_3	$I_{11}I_8$
9	I_9	$I_{10}I_{11}$	T_1	$I_{10}I_{11}$	$I_{11}I_{12}$	T_2	$I_{11}I_{12}$	$I_{12}I_{13}$	T_3	$I_{12}I_9$	I_9I_{10}	T_0	I_9I_{10}
10	I_{10}	$I_{12}I_{13}$	T_2	$I_{12}I_{13}$	$I_{13}I_{14}$	T_3	$I_{13}I_{10}$	$I_{10}I_{11}$	T_0	$I_{10}I_{11}$	$I_{11}I_{12}$	T_1	$I_{11}I_{12}$
11	I_{11}	$I_{14}I_{15}$	T_3	$I_{14}I_{11}$	$I_{11}I_{12}$	T_0	$I_{11}I_{12}$	$I_{12}I_{13}$	T_1	$I_{12}I_{13}$	$I_{13}I_{14}$	T_2	$I_{13}I_{14}$
12	I_{12}	$I_{12}I_{13}$	T_0	$I_{12}I_{13}$	$I_{13}I_{14}$	T_1	$I_{13}I_{14}$	$I_{14}I_{15}$	T_2	$I_{14}I_{15}$	$I_{15}I_0$	T_3	$I_{15}I_{12}$
13	I_{13}	$I_{14}I_{15}$	T_1	$I_{14}I_{15}$	$I_{15}I_0$	T_2	$I_{15}I_0$	I_0I_1	T_3	I_0I_{13}	$I_{13}I_{14}$	T_0	$I_{13}I_{14}$
14	I_{14}	I_0I_1	T_2	I_0I_1	I_1I_2	T_3	I_1I_{14}	$I_{14}I_{15}$	T_0	$I_{14}I_{15}$	$I_{15}I_0$	T_1	$I_{15}I_0$
15	I_{15}	I_2I_3	T_3	I_2I_{15}	$I_{15}I_0$	T_0	$I_{15}I_0$	I_0I_1	T_1	I_0I_1	I_1I_2	T_2	I_1I_2

Figure 9: Execution Trace N = 16, M = 4

```
procedure C1D_1(M)
{O(1) memory one dimensional convolution}
begin
  PAIRING(M);
  C1D := 0;
  for j :=0 to M-1 do
  begin
        B(p) := I(p); (igray(p) mod M = M -1 -j)
        C1D := C1D + A * T;
        SHIFT(A, -1, P);
        C := B; B := A; A := C; { interchange A and B}
        SHIFT(T, -1, M);
  end
end {of C1D_1}
```

Figure 10: $O(1)$ memory computation of C1D

4 TWO DIMENSIONAL CONVOLUTION

Assume that $P = N^2$ PEs are available. These may be viewed as an N × N array as described in Section 2. We use (i, j) to refer to the PE in position(i, j) of the N × N array. Thus, for the example of Figure 1, PE(0, 0) is PE 0, PE(2, 3) is PE 7, and PE(3, 3) is PE 6. The index of PE(i, j) is $gray(iN + j)$ if i is even and $gray(iN + N - 1 - j)$ if i is odd. This corresponds to the snake like row major interpretation. We assume that $I[i, j]$ is initially in the I register of PE(i, j). Further since N and M are assumed to be powers of 2, the N × N array may further be viewed as composed of N^2/M^2 arrays of size M × M (use the tiling of section 2). We assume that T is initially in the top left such array.

4.1 O(M) Memory

When O(M) memory is available, PE(i,j), $0 \le i < N,\ 0 \le j < N$ computes M one dimensional convolutions $S(q),\ 0 \le q < M$ defined as below

$$S(q) = \sum_{r=0}^{M-1} I((i, (j + r) \bmod N) * T(q, r)$$

Next, C2D is obtained by performing an adjacent sum operation along the columns of the N × N PE array. A high level description of the algorithm is given in Figure 11. The total number of unit routes is $M^2 + O(M + \log(N/M))$.

```
procedure C2D_M(N, M)
        {assumes O(M) memory per PE}
Step1:  Broadcast T to all M × M blocks in the N × N PE array
Step2:  Perform a data accumulation on I. For this operation, the N × N PE array is viewed
        as N independent hypercubes with each row forming one such hypercube.
        Following the operation, each PE contains the M I values it needs to compute its
        S(q)'s.
Step3:  Compute the S(q)'s. Each S(q) is a one dimensional convolution. However, the
        data accumulation step of the algorithm of Figure 4 may be omitted as the I values
        have already been accumulated in Step 2. To go from one S to another, the T values
        need to be shifted along the columns of each M × M block.
Step4:  Compute C2D(i, j) = $\sum_{r=0}^{M-1} S[r]((i+r) \bmod N, j)$. This is done using the adjacent sum
        algorithm of Section 2 on the columns of the N × N PE array
end
```

Figure 11: High level description of two dimensional convolution with each PE having O(M) Memory

4.2 O(1) Memory

Now, it is not possible for each PE to accumulate the M values of I it needs from its row. Nor is it possible for a PE to compute the values $S(q)$, $0 \le q < M$. We may rewrite the definition of C2D as

$$C2D[i, j] = \sum_{r=0}^{M-1} CXD[i, r, j]$$

where

$$CXD[i, r, j] = \sum_{a=0}^{M-1} I[(i+r) \bmod N, (j+a) \bmod N] * T[r, a]$$

Some of the CXD terms needed for the computation of $C2D(i, j)$ can be computed within the M × M PE block that contains $PE(i, j)$ as all the needed I and T values are in the block. The remaining terms can be computed by the corresponding PE in the next lower M × M block as this block contains the needed I values. Thus each PE computes an E value (for itself) and an F value (for the corresponding PE in the adjacent upper M × M block).

The E and F values are computed in k iterations. During iteration k, the PEs in the k'th row of each M × M PE block compute their E and F values. These rows have index k, $M+k$, $2M+k$, $\cdots$. Also

$$E(aM+k,\ j) = \sum_{r=0}^{M-1-k} CXD[aM+k,\ r,\ j] \text{ and}$$

$$F(aM+k,\ j) = \sum_{r=M-k}^{M-1} CXD[((a-1)M+k) \ mod\ N,\ r,\ j].$$

For this, we note that PE(i, j) is in the $i\ mod\ M$ row of the $\lfloor i/M \rfloor$'th M × M block. So, each PE needs to compute

$$A = CXD[\ \lfloor i/M \rfloor\ M + k,\ i\ mod\ M - k,\ j\ ] \text{ if } i\ mod\ M \geq k \text{ and}$$

$$B = CXD[\ \lfloor i/M \rfloor\ M + k - M,\ i\ mod\ M - k + M,\ j\ ] \text{ if } i\ mod\ M < k$$

Then, the PEs in rows $aM + k,\ 0 \leq a < N/M$ can compute E and F by summing the As and Bs in their column and in their M × M block. Once this has been done, C2D is computed by shifting the F's up the columns by M units and adding to the E's. A high level description of the algorithm is provided in Figure 12. The total number of unit routes is $2M^2 + O(M + \log(MN))$.

```
procedure C2D_1(N, M)
          {assumes O(1) memory per PE}
Step1:    Broadcast T to all M × M blocks in the N × N PE array
Step2:    Repeat Steps 3 and 4 for k := 0 to M-1
Step3:    PE(i, j) computes $CXD[(\lfloor \frac{i}{M} \rfloor M + k) mod N,\ i\ mod\ M - k,\ j\ ]$ if $i\ mod\ M \geq k$ using
          C1D_1(M) and puts the result in A, otherwise A = 0;
          PE(i, j) computes $CXD[(\lfloor \frac{i}{M} \rfloor M + k - M)\ mod\ N,\ i\ mod\ M - k + M,\ j\ ]$ if $i\ mod\ M < k$
          using C1D_1(M) and puts the result in B, otherwise B = 0;
Step4:    Use the data sum operation, described in Section 2, to sum the B's and A's in
          $PE(\lfloor \frac{i}{M} \rfloor M + k,\ j)$ in F and E respectively. Shift the T values up the columns by 1.
Step5:    SHIFT(F, -M, N) along columns . C2D := E + F .
```

Figure 12: High level description of two dimensional convolution with each PE having O(1) Memory

5 MEDIUM GRAIN TEMPLATE MATCHING

In the previous sections we have developed algorithms to perform template matching on a fine grain hypercube. Such a computer has the property that the cost of interprocessor communication is comparable to that of a basic arithmetic operation. In this section, we shall consider the template matching problem on a hypercube in which interprocessor communication is relatively expensive and the number of processors is small relative to the image size n. In particular we shall experiment with an NCUBE/7 hypercube which is capable of having up to 128 processors. The NCUBE/7 available to us, however, has only 64 processors. The time to perform a two byte integer addition on each hypercube processor is 4.3 microseconds whereas the time to communicate b bytes to a neighbor processor is approximately $447 + 2.4b$ microseconds.

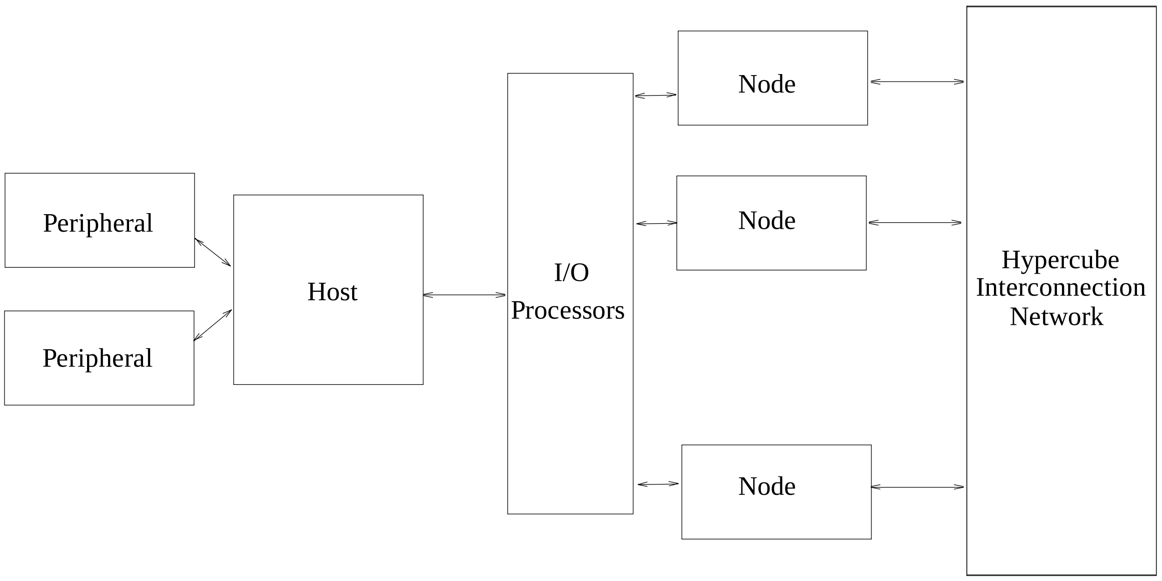

Figure 13: NCUBE hypercube computer

Figure 13 shows the block diagram for the NCUBE/7 hypercube multicomputer. Several cases of the template matching problem can be studied. These vary in the initial location of the image and the template and the final location of the convolution (result matrix). We consider the following cases . In all of these, the template is initially in the host.

1. Host-to-host: The image is in the host initially and the result is to be left in the host also.
2. Hypercube-to-host: The image is initially in the host but the result is left in the hypercube.
3. Hypercube-to-hypercube: The image is initially in the hypercube and the convolution is to be left there too.

Let p be the number of hypercube processors. We assume that p is a perfect square and that $\sqrt{p}$ divides n. Hence, the hypercube may be visualized as a $\sqrt{p} \times \sqrt{p}$ mesh and the $n \times n$ convolution matrix can be mapped onto this with each processor getting an $n/\sqrt{p} \times n/\sqrt{p}$ block. We assume that each processor has enough memory to hold one copy of the $m \times m$ template. As far as mapping the $n \times n$ image is concerned, we consider the two possibilities:

(1) Overlap Mapping: In this, each processor gets enough of the image to compute all its convolution values. Hence, the processor in position (0, 0) of the mesh gets I[$0 \,..\, n/\sqrt{p} + m - 2$, $0 \,..\, n/\sqrt{p} + m - 2$].

(2) Nonoverlap Mapping: The image is decomposed into $n/\sqrt{p} \times n/\sqrt{p}$ blocks. This is done in the same way as the convolution decomposition. Each processor gets the image block that corresponds to its convolution block.

Notice that if overlap mapping is used, then the host must transfer more data to each hypercube processor than when the nonoverlap mapping is used. However, no interprocessor communication is needed when the overlap mapping is used. Interprocessor communication is, however, needed when the nonoverlap mapping is used. This can take the form of each processor communicating to its north, east, and northeast neighbor processors the image values they need to compute their convolution. Alternatively, each processor can compute the partial convolution values for its north, northeast, and east neighbors and then communicate these values. In either case, the communication overhead is the same. In our programs, we adopt the latter strategy.

It is also important to note that the communication overhead in the template matching problem is small relative to the computing cost. When the overlap mapping is used, $O(nm\sqrt{p} + pm^2)$ additional data is transmitted from the host to the hypercube nodes (i.e., in addition to the transfer of n^2 image values). However since the host can send data to several nodes in parallel, the overhead penalty is not as severe. While the same amount of data has to be transferred between processors when the nonoverlap mapping is used, the p processors can work in parallel so that the transfer time is approximately that for the transfer of $O(nm/\sqrt{p} + m^2)$ data. In either case, this overhead is expected to be small compared to the time required for the $O(n^2m^2/p)$ computing to be done by each node.

In each of the three cases listed above, we have assumed that the host broadcasts the template to the hypercube processors using a tree expansion scheme.

The NCUBE/7 run times for $p = 1, 4, 16,$ and 64; $n = 32, 64, 128, 252,$ and 512 and $m = 4, 8, 16,$ and 32 for the overlap memory mapping are given in Figures 14 through 16. For smaller values of p, the template matching can be done only for small n as there isn't enough memory on a hypercube processor to hold the convolution and the image subblocks assigned to it.

		m			
p	n	4	8	16	32
1	32	0.456	1.479	5.391	20.439
	64	1.832	5.867	21.169	81.485
4	32	0.142	0.383	1.366	5.223
	64	0.524	1.480	5.392	20.440
	128	2.022	5.869	21.170	81.487
16	32	0.104	0.176	0.478	1.596
	64	0.238	0.507	1.477	5.225
	128	0.790	1.754	5.394	20.442
	256	2.925	6.592	21.173	81.491
64	32	0.270	0.421	0.910	2.590
	64	0.428	0.643	1.246	3.172
	128	0.933	1.273	2.349	7.029
	256	2.724	3.293	7.205	22.069
	512	9.365	10.597	25.243	81.491

Times are in seconds
m = template size
n = image size
p = number of processors
Figure 14: Overlap Mapping: Host-to-Host

The figures show that for the case $n = 512$, $m = 32$, and $p = 64$, the run times for the host-to-host case are approximately 2.6% higher than that for the hypercube-to-host case and approximately 13.0% higher than the hypercube-to-hypercube case. This reflects the cost of transmitting the image and the convolution between the host and the hypercube. The observed speed up is almost equal to the theoretical maximum of p. The speedup and efficiency ($speedup/p$) for $n = 64$ and $m = 8$ are shown in Figure 17.

		m			
p	n	4	8	16	32
1	32	0.407	1.308	4.773	18.200
	64	1.600	5.211	18.891	72.268
4	32	0.126	0.355	1.233	4.666
	64	0.462	1.364	4.860	18.226
	128	1.810	5.367	18.974	72.391
16	32	0.069	0.146	0.426	1.483
	64	0.198	0.456	1.402	5.022
	128	0.695	1.643	5.199	18.830
	256	2.620	6.279	19.875	73.350
64	32	0.108	0.190	0.459	1.424
	64	0.200	0.350	0.832	2.533
	128	0.511	0.880	2.111	6.539
	256	1.645	2.786	6.788	21.405
	512	5.968	9.831	24.341	79.440

Times are in seconds
Figure 15: Overlap Mapping: Hypercube-to-Host

		m			
p	n	4	8	16	32
1	32	0.376	1.274	4.727	18.134
	64	1.504	5.094	18.763	72.105
4	32	0.096	0.320	1.184	4.572
	64	0.378	1.275	4.729	18.136
	128	1.506	5.096	18.764	72.107
16	32	0.028	0.084	0.299	1.146
	64	0.098	0.322	1.185	4.573
	128	0.380	1.277	4.731	18.138
	256	1.508	5.097	18.767	72.109
64	32	0.013	0.027	0.086	0.291
	64	0.030	0.086	0.301	1.148
	128	0.100	0.324	1.187	4.575
	256	0.381	1.279	4.733	18.139
	512	1.510	5.099	18.768	72.110

Times are in seconds
Figure 16: Overlap Mapping: Hypercube-to-Hypercube

The run times for the nonoverlap mapping are presented only for the hypercube-to-hypercube case. In this case, there are two possibilities:

1. Overlap of computation and communication between nodes

	p	1	4	16	64
Host-to-host	Speedup	1.00	3.96	11.57	9.12
	Efficiency	1.00	0.99	0.72	0.14
Hypercube-to-host	Speedup	1.00	3.82	11.43	14.89
	Efficiency	1.00	0.95	0.71	0.23
Hypercube-to-hypercube	Speedup	1.00	3.99	15.82	59.23
	Efficiency	1.00	0.998	0.99	0.93

Times are in seconds
Figure 17: Overlap Mapping: Speedup and Efficiency for $n = 64$ and $m = 8$

2. No overlap of computation and communication between nodes

Our experiments indicate that there is no substantial difference in the run times in the above two cases. This is because the amount of computation is much larger than the amount of communication between nodes. The run times for the nonoverlap mapping are given in Figure 18. For small template sizes the nonoverlap method is significantly slower than the overlap method. For larger template sizes the difference in run time is not so significant. Much of the difference in the run time is attributable to the following observations:

1. The program for the nonoverlap case is considerably more complex and so has greater overhead than that for the overlap case.
2. The data transfer rate from the host to the nodes is much higher than that between nodes.

Figure 19 shows the time required by a Cray-2 supercomputer to perform template matching. These are approximately one fifth of the hypercube-to-hypercube times on the NCUBE/7 with 64 processors.

6 CONCLUSIONS

In this paper, we have presented simple and efficient algorithms for 1-D convolution and image template matching (2-D Convolution) on an MIMD hypercube multicomputer. Also, we have experimented with a 64 processor NCUBE hypercube and found that this computer can perform template matchings for large images and templates in about five times the time needed by the Cray-2 supercomputer. Thus, the NCUBE has a very good cost-performance ratio for this

		m			
p	n	4	8	16	32
1	32	0.505	1.857	7.000	20.450
4	32	0.139	0.482	1.417	
	64	0.514	1.872	7.026	20.497
16	32	0.045	0.115		
	64	0.142	0.484	1.422	
	128	0.516	1.874	7.031	20.510
64	32	0.021			
	64	0.047	0.118		
	128	0.144	0.487	1.426	
	256	0.519	1.878	7.036	20.520

Times are in seconds
Figure 18: Nonoverlap Mapping: Hypercube-to-Hypercube

problem.

	m			
n	4	8	16	32
64	0.007	0.023	0.086	0.345
128	0.022	0.080	0.300	1.205
256	0.073	0.283	1.118	4.485
512	0.273	1.082	4.273	17.350

Times are in seconds
Figure 19: Template Matching on Cray-2

7 REFERENCES

[BALL85] D. H. Ballard and C. M. Brown, "*Computer Vision*", **1985**, Prentice Hall, New Jersey.

[CHAN86] T. E. Chan and Y. Saad, "Multigrid algorithms on hypercube multiprocessor", *IEEE Transactions on Computers",* **Nov. 86**, pp 969-977.

[CHAN87] J. H. Chang, O. Ibarra, T. C. Pong, and S. Sohn, "Convolution on a Pyramid Computer", *International Conference on Parallel Processing*, **1987**, pp 780-782.

[DEKE81] E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM*

Journal on computing, **1981**, pp. 657-675.

[FANG85] Z. Fang, X. Li and L. M. Ni, "Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers", *IEEE CAPAMI workshop,* **1985**, pp 33-40.

[FANG86] Z. Fang and L. M. Ni, "Parallel Algorithms for 2-D convolution", *International Conference on Parallel Processing,* **1986**, pp 262-269.

[HORO85] E. Horowitz and S. Sahni, "*Fundamentals of Data Structures in Pascal*", Computer Science Press, **1985**.

[KUNG82] H. T. Kung and S. W. Song, "A Systolic 2-D Convolution Chip", *Multicomputers and Image Processing: Algorithms and Programs, editors: Preston and Uhr (Academic Press, New York),* **1982**, pp 373-384.

[LEE87] S. Y. Lee and J. K. Aggarwal, "Parallel 2-D convolution on a mesh connected array processor", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **July 1987**, pp 590-594.

[MARE86] M. Maresca and H.Li, "Morphological Operations on Mesh-connected Architecture : A generalized convolution Algorithm", *Proceedings of 1986 IEEE Computer Society Workshop on Computer Vision and Pattern Recognition* ,**1986**, pp 299-304.

[PRAS87] V. K. Prasanna Kumar and V. Krishnan, "Efficient Image Template Matching on SIMD Hypercube Machines", *International Conference on Parallel Processing,* **1987**, pp 765-771.

[RANK88a] S. Ranka and S. Sahni, "Convolution on an SIMD mesh-connected computer", *Proceedings 1988 International Conference on Parallel Processing*, Vol III, Algorithms and Applications, Penn State University Press, pp. 212-217.

[RANK88b] S. Ranka and S. Sahni, "Image Template Matching on SIMD hypercube multicomputers", *Proceedings 1988 International Conference on Parallel Processing*, Vol III, Algorithms and Applications, Penn State University Press, pp. 84-91.

[ROSE82] A. Rosenfeld and A. C. Kak, "*Digital Picture Processing*", Academic Press, **1982**

[THOM77] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer", *Communications of the ACM*, **1977**, pp 263-271.