Optimizations of Array Convolutions for SIMD Architectures

Joydip Kundu
Janice E. Cuny

COINS Technical Report 91-65
Novemeber 1991

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

This work was partially supported by the Office of Naval Research under contract N00014-89-J-1492.

Abstract

Communication overhead can easily offset performance increases due to massive parallelism. In this paper, we introduce code optimizations for SIMD architectures that reduce communication costs for array convolutions, an important class of array manipulations. Our techniques use a graph-based approach that is independent of the underlying problem and machine architectures.

1 Introduction

Communication overhead can easily offset performance increases due to massive parallelism. The overhead is particularly significant for fine-grained, SIMD architectures since relatively little computation is performed between successive communications and all processes are delayed by communications. In this paper, we introduce code optimizations for SIMD architectures that reduce communication costs for array convolutions, an important class of array manipulations.

Our work began as an adaptation for the Connection Machine of algebraic optimization techniques developed by Fisher and Highnam [5]; their techniques are reviewed in the next section. In Section 3, we describe our more general work which uses a graph-based approach that is independent of the underlying problem and machine architectures. In Section 4, we give examples of code produced by our optimizations and compare our results to those of Fisher and Highnam. In Section 5, we discuss limitations and extensions to our work and, in Section 6, we relate our work to that of others. In the final section, we present our conclusions.

2 Previous Work: Fisher and Highnam

Both our work and that of Fisher and Highnam develop heuristics for minimizing communication costs in array convolutions. We assume that such convolutions are specified as *macro instructions* which describe the computation at each point of the array in terms of a relative neighborhood of values. Thus, for example, a smoothing operation common to many vision algorithms, might be specified using relative offsets as

```
val = a*([.][.]val + [.][.+1]val + [.+1][.]val + [.][.-1]val
  + [.-1][.]val + [.+1][.+1]val + [.+1][.-1]val + [.-1][.-1]val
  + [.-1][.+1]val);
```

and represented spatially by the mask

1	1	1
1	1	1
1	1	1

Fisher and Highnam's heuristics used macro instructions specified in terms of the directionals L, R, U, and D (with the obvious interpretation of left, right, up, and down respectively). Thus they would input the above instruction as

```
val = a*(val + U val + D val + L val + R val + RU val + RD val
      + LD val + LU val );
```

Their heuristics are based on an algebra that allows them to manipulate directional expressions in performing common subexpression elimination. For the smoothing example, they produce the following, optimal code

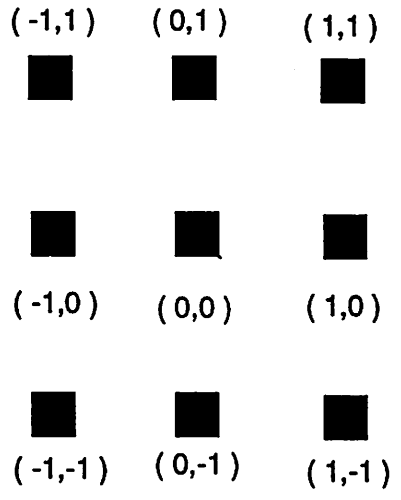
```
        t1 = val + L val + R val;
        t2 = val + U t1 + D t1;
        val = a*t2;
```

which reduces the number of both communication and addition steps from 8 to 4.

In adapting their techniques for the Connection Machine, we attempted to address a limitation: the reliance on a directional algebra that applied only to grids. Their heuristics assumed that both the problem and the machine architectures were grids and that the machine architecture was of dimension less than or equal to that of the input. This meant that they are unable to fully exploit the interconnectivity of an architecture such as the Connection Machine. Our heuristics eliminate these restrictions and additionally allow structures with nongrid topologies.

3 Our Approach

We call the points in the mask of a macro instruction the *communication space* of that instruction. The point on the left hand side of the equation is called the *destination* node and those on the right hand side are called *source* nodes. Thus, labeling the points in the the smoothing example above by their relative position from the destination node, we get the communication space



where the destination point is (0,0) and the source points are (-1,1), (0,1), (1,1), (-1,0), (0,0), (1,0), (-1,-1), (0,-1), and (1,-1).

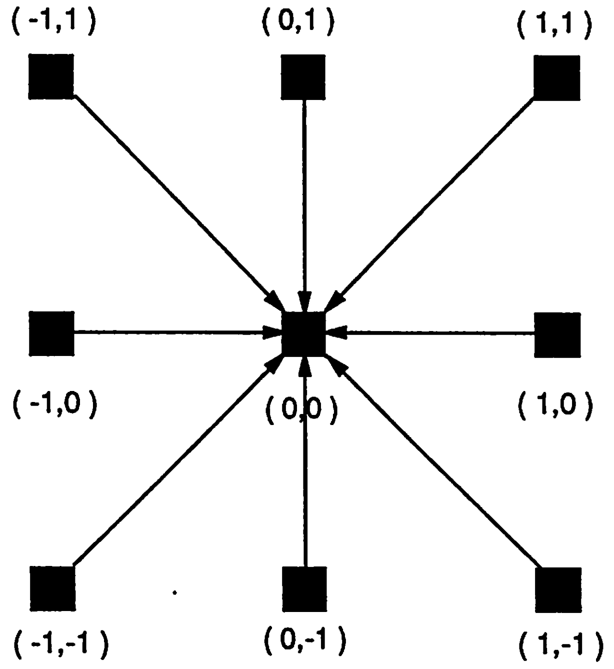
On the Connection Machine, macro instructions are implemented by a sequence of *micro instructions.* For the smoothing example, the C* compiler produced the micro instructions

```
t1 = [.][.]val;
t2 = [.+1][.]val;
t1 += t2;
t2 = [.-1][.]val;
t1 += t2;
t2 = [.][.+1]val;
t1 += t2;
```

```
t2 = [.][.-1]val;
t1 += t2;
t2 = [.+1][.+1]val;
t1 += t2;
t2 = [.+1][.-1]val;
t1 += t2;
t2 = [.-1][.+1]val;
t1 += t2;
t2 = [.-1][.-1]val;
t1 += t2;
[.][.]val = a* t1;
```

which contains two different types of micro instructions, one denoting communication and the other denoting local computation.

If we represent the communication instructions in this example by directed edges in the communication space, we get the following graph:



Any directed graph that connects the communication space of a macro instruction such that one and only one path exists from every source node to the destination node is called an *instruction graph.* The degree of each vertex in such a graph represents the number of local computations to be performed at that vertex.

Every sequence of micro instructions corresponds to a single instruction graph. There may, however, be many different sequences of micro instructions — and hence different instruction graphs — for a single macro instruction. Our objective is to find an instruction graph with minimal cost, using some *distance* metric, and to use that instruction graph in generating efficient code. The process of finding the minimal cost graph can also be thought of as finding the minimum spanning tree (MST) rooted at the destination node with respect to the graph completely connecting the communication space. For the remainder of this report, whenever we refer to MST we will be referring to the instruction graph with the minimum cost.

In determining the cost of an instruction graph, we label each edge with a weight according to our distance metric and then take the sum of those weights over the graph. Our distance metric was chosen to reflect the characteristics of the Connection Machine (specifically the CM-2) but other metrics could easily be substituted without altering our basic heuristics.

For a fully configured CM-2 system, the interconnection network is a 12-cube connecting 4096 processor chips. Each router node is connected to 12 other router nodes; specifically, router node i is connected to router node j if and only if $|i - j| = 2^k = (j \oplus k) \vee 2^k$. In other words, routers i and j are connected along dimension k if and only if i and j differ only in bit position k.

In overlaying a grid on a hypercube, we use Grey-coding in which the points of the grid are mapped to hypercube addresses in such a way that adjacent grid points have labels that differ in exactly one bit position. Thus adjacent grid nodes will be connected by ypercube wires. For example, the Grey distance between two points P(x_1,y_1,z_1) and Q(x_2,y_2,z_2) in a 3 dimensional grid is given by:

Grey distance = weight $(|x_1 - x_2|)$ + weight $(|y_1 - y_2|)$
+ weight $(|z_1 - z_2|)$,

where x_1, y_1, z_1 and x_2, y_2, z_2 are expressed in Grey code, and

weight(a) = # of bits that are 1 in the grey representation of a.

The Grey distance is our distance metric.

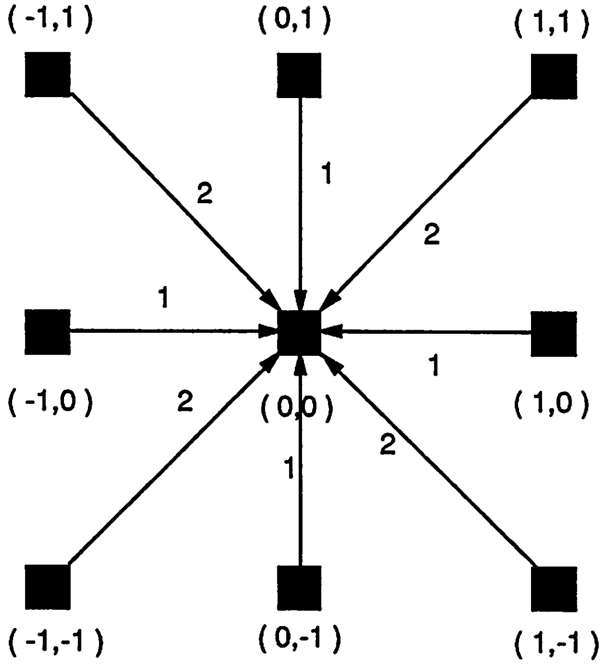


Figure 1: Instruction graph for 3x3 smoothing operation.

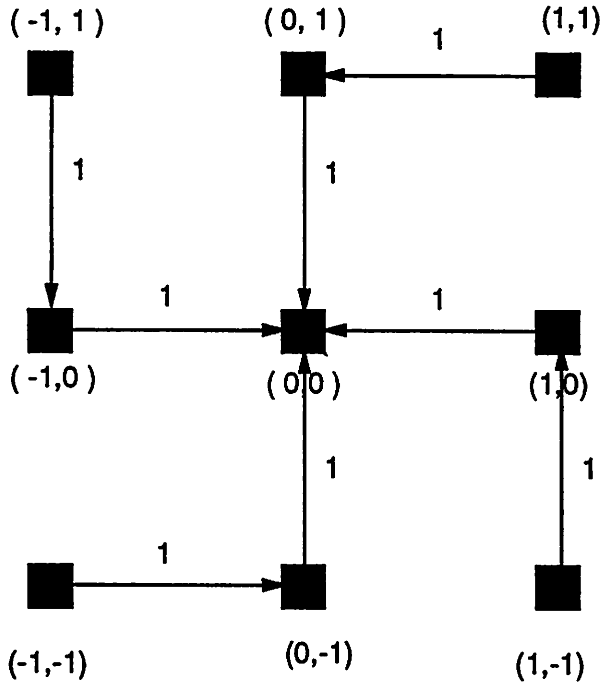


Figure 2: Instruction graph with minimum cost for 3x3 smoothing operation.

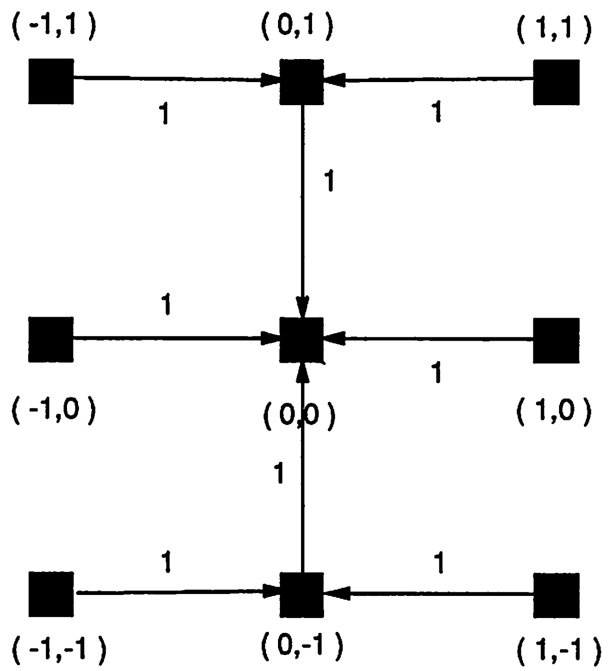


Figure 3: Alternative instruction graph with minimum cost for 3x3 smoothing operation.

To compute the cost of an instruction graph we label each edge with the Grey distance between its end points and take the sum of all the edge weights. The instruction graph above has a cost of 12 as shown in Figure 1. A minimal cost graph for the same macro instruction has a cost of 8; examples are shown in Figures 2 and 3.

Not all minimum cost instruction graphs, however, are equally good. The micro instruction sequence corresponding to the graph in Figure 2 is

```
t1 = [.+1][.]val;
t1 += val;
t2 = [.][.+1]t1;
t2 += val;
t1 = [.][.+1]val;
t1 += val;
```

```
t3 = [.-1][.]t1;
t2 += t3;
t1 = [.][.-1]val;
t1 += val;
t3 = [.][.-1]t1;
t2 += t3;
t1 = [.-1][.]val;
t1 += val;
t3 = [.][.+1]t1;
t2 += t3;
[.][.]val = a*t2;
```

The graph in Figure 3, admits more optimization. In generating the code for this graph, we note that the points (1,1), (1,0) and (1,-1) are equivalent, in the sense that each of them holds the same variable (*val*) and each is in the same relative position its parent ([.-1][.]). Thus after the single micro instruction,

$$t_1 = [. + 1][.]val$$

all three nodes have sent data to their parents. Similarly, the points (-1,1), (-1,0) and (-1,-1) are also equivalent and all send their data to their parents with the micro instruction

$$t_2 = [. - 1][.]val$$

Exploiting this fact, we can use a single micro instruction to send data from equivalent points to their parents and optimize the code to:

```
t1 = [.+1][.]val;
t1 += [.][.]val;
t2 = [.-1][.]val;
t1 += t2;
t2 = [.][.+1]t1;
t2 += [.][.]t1;
t3 = [.][.-1]t1;
t2 += t3;
[.][.]val = a*t2;
```

requiring 4 communication steps and 4 local additions, matching the Fisher and Highnam optimization.

In general, we want to find a MST such that a directed edge from node i to node j exists if and only if

> j $\in$ {K} where K is the set of nodes that are closest to node i, and j is the node closest to the destination node $\forall$j $\in$ {K}, and where the distance metric used is the Grey coded distance.

Our stand-alone program implicitly builds such a minimum spanning tree, and generates a set of micro instructions for any given macro communication instruction.

Before presenting our heuristic, we need several definitions. Given a location in the communication space, we define the *active variables* at that location to be those variables holding the values of subexpressions still needed in the computation. In addition, given nodes n_1 and n_2, we define the set of nodes *similar* to node n_1 with respect to node n_2 to be that set of nodes e_1 such that

i. there is a node e_2 closer to the destination node than e_1, and

ii. both n_1 and e_1 can transfer their active variables to n_2 and e_2 respectively with the same micro instruction.

Procedure for Code Generation

The input to our algorithm is a set of nodes together with pointers to their active variables.

1. Sort the points in the communication space in decreasing order according to their Grey distances from the destination node.

2. Build the implicit MST as follows:

   Choose the first node of the list, n_1 and find its closest neighbor, n_2. If there is more than one closest neighbor, choose the node n_2 such that there are the greatest number of nodes similar to n_1 with respect to n_2. Generate an instruction to send the active variable of node n_1 to node n_2, remove n_1 and all nodes similar to n_1 with respect to n_2 from the list, and update the active variables accordingly.

   Repeat until the list of unconnected nodes is empty.

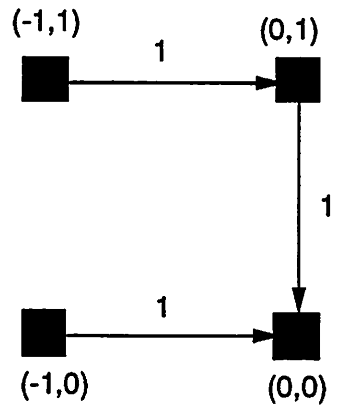


Figure 4: Instruction graph for 2x2 summation.

4 Examples

In this section we give some examples of our optimizations. We implemented the optimizer with the intention of validating our heuristics and thus, we have not bothered to include other obvious optimizations that could be performed with standard techniques. Thus for example, our code often contains unnecessary register transfers near the end of instruction sequences; in our counts of temporary variables we include in parenthesis the number needed if we were to incorporate standard live and dead variable analysis (as in [6]) to avoid this. In addition because neighbor to neighbor communication on the Connection Machine is faster than arbitrary communication, we have included counts of both the number of communication instructions and the number cube dimensions traversed by those instructions (communication hops).

1. Optimization Example (1): 2x2 summation (Figure 4)

```
Input:
[.][.]p = [.][.]p + [.][.+1]p + [.-1][.]p + [.-1][.+1]p;

Equivalent Connection Machine Code from C* Compiler:
      t1 = [.][.+1]p;
```

```
        t2 = [.][.]p;
        t1 += t2;
        t2 = [.-1][.]p;
        t1 += t2;
        t2 = [.-1][.+1]p;
        t1 += t2;
        [.][.]p = t1;

# of communication instructions = 4
# of communication hops = 4
# of local computations =  4
# of temporary variables = 2

Optimized Output:
        t1 = [.-1][.]p;
        t1 += p;
        t2 = [.][.+1]t1;
        t3 = t1 + t2;
        t1 = t3
        [.][.]p = t1;

# of communication instructions = 2
# of communication hops = 2
# of local computations =  4
# of temporary variables = 3 (2)
```

2. Optimization Example (2): 3x3 Symmetric Filter (Figure 5)

```
Input:
[.][.]result = a*[.-1][.+1]p + b*[.][.+1]p + a*[.+1][.+1]p +
         b*[.-1][.]p + c*[.][.]p + b*[.+1][.]p +
         a*[.-1][.-1]p + b*[.][.-1]p + a*[.+1][.-1]p;

Equivalent Connection Machine Code from C* Compiler:
        t1 = [.-1][.+1]p;
```

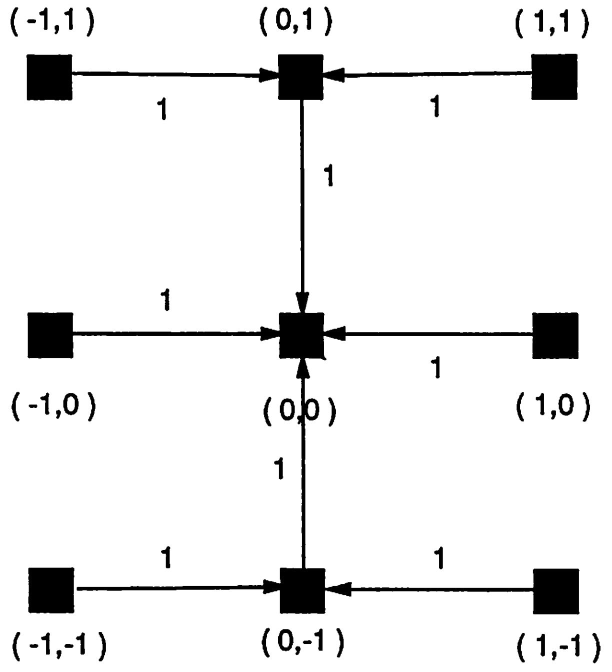


Figure 5: Instruction graph for 3x3 symmetric filter.

```
        t1 = a*t1;
        t2 = [.][.+1]p;
        t2 = b*t2;
        t1 += t2;
        t2 = [.+1][.+1]p;
        t2 = a*t3;
        t1 += t2;
        t2 = [.-1][]p;
        t2 = b*t2;
        t1 += t2;
        t2 = [.][.]p;
        t2 = c*t2;
        t1 += t2;
        t2 = [.+1][.]p;
        t2 = b*t2;
        t1 += t2;
        t2 = [.-1][.-1]p;
        t2 = a*t2;
        t1 += t2;
        t2 =  [.][.-1]p;
        t2 = b*t2;
        t1 += t2;
        t2 = [.+1][.-1]p;
        t2 = a*t2;
        t1 += t2;
        [.][.]result = t1;

# of communication instructions = 9
# of communication hops = 12
# of local computations = 18
# of temporary variables = 2


Optimized Code:
        t1 = c * p;
        t2 = b * p;
        t3 = a * p;
        t4 = [.-1][.]t3;
        t5 = t2 + t4;
        t6 = [.+1][.]t3;
        t5 += t6;
        t7 = 0;
        t8 = [.-1][.]t2;
        t7 += t8;
        t9 = [.+1][.]t2;
        t7 += t9;
        t10 = [.][.-1]t5;
        t7 += t10;
        t11 = [.][.+1]t5;
        t7 += t11;
        t12 = t1;
        t7 += t4
        t4 = t7
        [.][.]result = t4;

# of communication instructions = 6
# of communication hops = 6
# of local computations = 14
# of temporary variables  = 12 (7)
```

In all the above examples our optimizer generates the same optimal set of instructions as the Fisher/Highnam algebraic approach because one hop neighbor to neighbor communications are always available on the grid and extra cube connections are superfluous. This is not always the case as we show in the next two examples where our optimizer generates code using fewer communication hops than that generated by the the algebraic approach.

1. Mask A: (Figure 6)

```
Input:
[.][.]p = [.][.]p  + [.][.+1]p  + [.][.-1]p + [.+1][.]p
         + [.-1][.]p + [.+1][.+1]p +  [.+1][.-1]p + [.-1][.+1]p
         + [.-1][.-1]p + [.][.+3]p + [.][.-3]p;
```

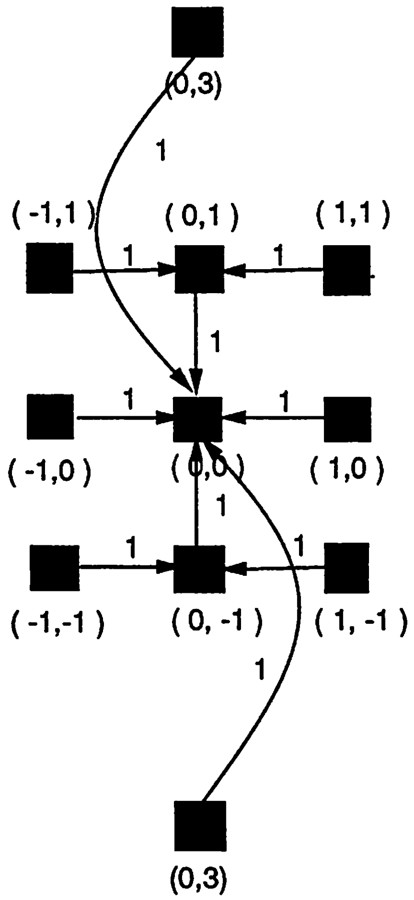


Figure 6: Instruction graph for mask A.

```
Algebraic Optimizer Output:
        t1 = UUp;
        t2 = DDp;
        t3 = p + Lp + Rp;
        t4 = t1 + t3;
        t5 = t2 + t3;
        t6 = t3 + Ut4 + Dt5;
        p = t6;

# of communication instructions = 6
# of communication hops = 8
# of local computations = 10
# of temporary variables = 6

Equivalent Connection Machine Code from C* Compiler:
        t1 = p;
        t2 = [.][.-1]p;
        t1 += t2;
        t2 = [.][.+1]p;
        t1 += t2;
        t2 = [.+1][.]p;
        t1 += t2;
        t1 = [.-1][.]p;
        t1 += t2;
        t2 = [.+1][.+1]p;
        t1 += t2;
        t2 = [.+1][.-1]p;
        t1 += t2;
        t2 = [.-1][.+1]p;
        t1 += t2;
        t2 = [.-1][.-1]p;
        t1 += t2;
        t2 = [.][.+3]p;
        t1 += t2;
        t2 = [.][.-3]p;
        t1 += t2;
        [.][.]my = t1;
```

```
# of communication instructions = 11
# of communication steps = 14
# of local computations =  13
# of temporary variables = 2
```

```
Our Optimizer Output:

       t1 = [.-1][.]p;
       t1 += p;
       t2 = [.+1][.]p;
       t1 += t2;
       t3 = [.][.-3]p;
       t4 = t1 + t3;
       t5 = [.][.-1]t1;
       t4 += t5;
       t6 = [.][.+3]p;
       t4 += t6;
       t7 = [.][.+1]t1;
       t4 += t7;
       t1 = t4
       [.][.]p = t1;

# of communication instructions = 6
# of communication hops = 6
# of local computations = 8
# of temporary variables = 7
```

The optimizer based on algebraic approach misses the cube path of distance 1 available from both the nodes (0,3) and (0,-3) to the origin.

2. Mask B (Figure 7)

```
  Input : [.][.]p = [.][.+3]p + [.+3][.+1]p + [.+7][.+3]p;

 Algebraic optimizer Output:
```

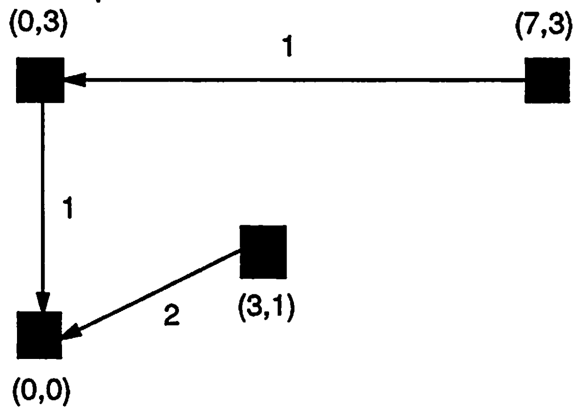


Figure 7: Instruction graph for mask B.

```
    t1 = RRRRUUp;
    t1 += p;
    t2 = RRRUt1;
    t2 += p;
    t3 = UUUp;
    t2 += t3;
     p = t2;

# of communication instructions = 3
# of communication hops = 10 ( 7 if we assume cube paths are used )
# of local computations =  3
# of temporaries = 3

  Equivalent Connection Machine Code from C* Compiler:
       t1 = p;
       t2 = [.][.+3]p;
       t1 += t2;
       t2 = [.+3][.+1]p;
       t1 += t2;
       t2 = [.+7][.+3]p;
       t1 += t2;
```

Comparison between current C*, Algebraic and MST based Optimizations												
Examples	# of instr			# of hops			# of temps			# of local comps		
	C*	Alg	MST	C*	Alg	MST	C*	Alg	MST	C*	Alg	MST
3x3 filter	9	6	6	12	6	6	2	7	7	18	14	14
2x2 sum	4	2	2	4	2	2	2	2	2	4	4	4
Mask A	11	6	6	14	8	6	2	6	7	13	10	8
Mask B	3	3	3	5	7	3	4	3	3	3	3	3

Figure 8: Summary of performance of different optimization techniques

```
        [.][.]p = t1;

# of communication instructions = 3
# of communication hops = 5
# of local computations =  3
# of temporaries = 2

 Our Optimizer Output:
        t1 = [.+7][.]p;
        t1 += p;
        t2 = [.+3][.+1]p
        t3 = [.][.+3]t1;
        t3 += t2;
        p  = t3;
# of communication instructions = 3
# of communication hops =  4
# of local computations =  3
# of temporaries = 3 (2)
```

In this case too, the algebraic approach fails to see that there is a cube path of distance 1 available between the nodes (7,3) and (0,3) which is shorter than the grey distance between nodes (7,3) and (3,1).

5 Caveats

As discussed above, our original goal was to explain the theory behind our optimization and show that a practical peephole optimizer can be built around that theory. Thus there were a number of issues that we did not address.

We did not address the issue of global optimization. Consider the code

```
Input:
[.][.]gx = b*(y*[.+1][.+1]p + x*[.][.+1]p - y*[.-1][.+1]p )
        + a*(y*[.+1][.]p + x*[.][.]p - y*[.-1][.]p )
        + b*(y*[.+1][.-1]p + x*[.][.-1]p - y*[.-1][.-1]p );
```

Our optimizer produced the following micro instructions (having the instruction graph of 3, but interpreting it as four different instruction graphs due to the higher precedence associated with the parentheses)

```
t1 = x * p;
t2 = y * p;
t3 = [.-1][.]t2;
t1 -= t3;
t4 = [.+1][.]t2;
t1 += t4;
t3 = t1
t4 = x * p;
t5 = y * p;
t6 = [.-1][.]t5;
t4 -= t6;
t7 = [.+1][.]t5;
t4 += t7;
t6 = t4
t7 = x * p;
t4 += t7;
t6 = t4
t7 = x * p;
t8 = y * p;
t9 = [.-1][.]t8;
t7 -= t9;
t10 = [.+1][.]t8;
```

```
t7 += t10;
t9 = t7
t10 = a * t6;
t11 = b * t9;
t12 = 0;
t13 = [.][.-1]t11;
t12 += t13;
t14 = [.][.+1]t10;
t12 += t14;
t15 = t10;
t12 += t15;
[.][.]gx = t12;

# of communication instructions = 8
# of communication hops = 8
# of local computations = 26
# of temporary variables = 15
```

If, however, we had used standard live and dead variable analysis we could have generated the code

```
t1 = x*p;
t2 = y*p;
t3 = [.-1][.]t2;
t1 -= t3;
t4 = [.+1][.]t2;
t1 += t4;
t3 = a*t1;
t4 = b*t1;
t5 = [.][.+1]t4;
t3 += t5;
t5 = [.][.-1]t4;
t3 += t5;
[.][.]gx = t3;

# of communication instructions = 4
# of communication hops = 4
# of local computations = 9
# of temporary variables = 5
```

(with the instruction graph of Figure 3). The failure of the optimizer is due to the fact that our compiler considers expressions within parenthesis to be of higher precedence than the ones outside. If there are multiple expressions delimited by parentheses, they are treated as separate macro instructions.

Other limitations of our stand-alone optimizer are

- The scanner handles only integer literals.
- A variable name can only contain uppercase or lowercase letters. Numerals are not allowed.
- Only one local computation is allowed. That is, we can have expressions like $a * [.+2][.+4]p$ and $a * (b * [.+2][.+4]p)$ but not $a * b * [.+2][.+4]p$

All of these limitations could be removed quite easily but that work is beyond the scope of this paper.

6 Comparisons to Other Work

The move from low level programming languages to higher level languages supporting array operations will require the development of efficient mechanisms for data distribution and alignment. Much of the existing work [3, 9, 10, 12, 14] has been concerned with reducing the need for data movement by considering patterns of reference in initial allocation decisions. Our work seeks to minimize data motion given these original allocations.

Other authors addressing similar issues of post-allocation data movement include Albert and Knobe, *et. al* [3, 9] and Gilbert [7]. Albert and Knobe *et. al.* briefly mention two types of common subexpression elimination (cse) for reducing communication: standard cse and a new *generalized cse*. The later of these is very much like the work by Fisher and Highnam. The former relies on existing techniques but, it should be noted, does not necessarily improve performance in our domain. In an example from their paper[1], consider an implementation of the macro instruction

[1]Example adapted from [3], page 55.

$$val = [.][. + 1]val + [.][. - 1]val + [. - 1][.]val + [. + 1][.]val$$

where each process sums the values from its four immediate neighbors. Many pairs of values get summed together at more than one location: the values in locations [3][7] and [4][6], for example, are summed together in computing the value for [4][7] and again in computing the value for [3][6]. The authors suggest computing their sum once at [3][6] and sending the result of that computation to [4][7]. This reduces the number of necessary communication steps from 4 to 3 but does not necessarily reduce the number of communication hops. On the CM, the diagonal transmission (from [3][6] to [4][7]) might well require more time than the two direct hops it replaces.

Gilbert [7] also presents an algorithm to minimize code motion subsequent to array allocation. His algorithm is considerably more complex than ours but it allows an additional degree of freedom in the solution: intermediate results can be calculated at nodes not in the original communication space of the instruction.

Finally, related work has been done on optimizing the performance of array convolutions for programs in which more than one array point is allocated to a process [4, 13]. Our techniques could be used in conjunction with these optimizations but as the number of points allocated to processes increases the effectiveness of our optimizations will decrease.

7 Conclusions

In our effort to adapt the directional algebraic approach to the Connection Machine architecture, we have developed a new graph-theoretic approach to optimization which is more general. Our approach has eliminated the dependence on the problem architecture: if the architecture of the problem or the machine changes, we need only change the distance metric. In the examples shown above, where the extra CM connections are not needed, our optimizer produces the same optimized code as the directional algebraic technique; where the extra cube paths can be exploited, our optimizer produces better code.

References

[1] *Programming in C/Paris*, The Connection Machine Manual. Thinking Machines Corporation, Cambridge, Massachusetts, 1989.

[2] *Programming in C**, The Connection Machine Manual. Thinking Machines Corporation, Cambridge, Massachusetts, 1990.

[3] Eugene Albert, Kathleen Knobe, Joan D. Lucas, and Guy L. Steele Jr., "Compiling Fortran 8x Array Features for the Connection Machine Computer System," *Proceedings ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications*, pp. 42-56 (1988).

[4] Mark Bromley, Steven Heller, Tim McNerney, and Guy teele Jr, "Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler,"*SIGPLAN Notices*, 26(6), pp. 145-156 (June 1991).

[5] A. L. Fisher and P. T. Highnam. "Communication and code optimization in SIMD programs," *International Conference on Parallel Processing,* 1989, pp. 84-88.

[6] C. N. Fisher and R. J. Leblanc. *Crafting a compiler.* The Benjamin/Cummings Publishing Company, Inc.

[7] John R. Gilbert and Robert Schreiber, "Optimal Expression Evaluation for Data Parallel Architectures," JPDC 13, pp. 58-64 (1991).

[8] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Massachusetts, 1985.

[9] Kathleen Knobe, Joan D. Lucas, and Guy L. Steele Jr., "Massively Parallel Data Optimization", *Proceedings 1988 Second Symposium on the Frontiers of Massively Parallel Computations*, pp.551-558 (1988).

[10] Kathleen Knobe, Joan D. Lucas, and Guy L. Steele Jr., "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines", JPDC 8(2), pp. 102-118 (1990).

[11] Kathleen Knobe and Venkataraman Natarajan, "Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD Machines,"

Proceedings Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation, pp. 416-423 (October 1991).

[12] J. Li, and M. Chen, "Index Domain Alignment: Minimizing Cost of Crossreferencing between Distributed Arrays," *Proceedings Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pp. 424-433 (October 1991).

[13] Jacek Myczkowski and Guy Steele. "Seismic Modeling at 14 Gigaflops on the Connection Machine," *Supercomputing 91.*

[14] J. F. Prins, "A Framework for Efficient Execution of Array-Based Languages on SIMD Computers," *Proceedings Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pp. 462-470 (October 1991).