Loop Rotation

Michael Wolfe

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-004
May, 1989

This work was supported by Defense Advanced Research Projects Agency under grant no. MDA972-88-J-1004.

Loop Rotation

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-004
May, 1989

Loop Rotation

Michael Wolfe
Oregon Graduate Center
19600 NW von Neumann Drive
Beaverton, OR 97006

mwolfe@cse.ogc.edu
503-690-1153

Abstract

We want to map shared-memory programs, such as Fortran programs (perhaps with parallel syntax), onto a distributed memory message passing multiprocessor (DMMP), such as a hypercube. Using a domain decomposition approach, two things need to be specified: the partitioning of the data among the processors and the communication between the processors. We assume that the user will partition the data using language extensions and the compiler will insert communication between processors. We show how parallel loops can be mapped into distributed code by prefetching all required data before doing any computation, which can introduce broadcast operations for certain types of loops, which we call *systolic loops*. To remove the broadcast, we use variants of the wavefront method, at a loss of efficiency. We then introduce a new program transformation, called *loop rotation*, specifically targeted for mapping systolic loops onto DMMPs. When it applies, loop rotation produces an efficient program with uniform communication path utilization. We then discuss how to apply loop rotation in a three-dimensional loop, where it improves efficiency but does not produce a perfectly parallel algorithm. Finally, we use loop rotation as a mechanism to compile code for which the data was distributed in a block-diagonal or block-anti-diagonal manner.

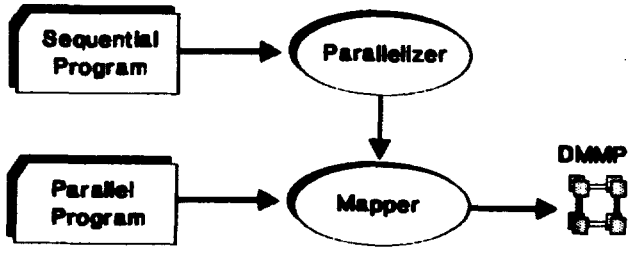
Loop Rotation

Michael Wolfe
Oregon Graduate Center
19600 NW von Neumann Drive
Beaverton, OR 97006

mwolfe@cse.ogc.edu
503-690-1153

1. Prologue

We want to map shared-memory programs, such as Fortran programs (perhaps with parallel syntax), onto a distributed memory message passing multiprocessor (DMMP), such as a hypercube. We distinguish automatic discovery of parallelism in sequential programs from mapping parallel shared-memory programs onto DMMPs. This paper is concerned with the mapping phase. Thus a sequential program can be automatically (or semi-automatically) converted to parallel form and then mapped onto a DMMP, while a parallel program can proceed directly to the mapping phase:

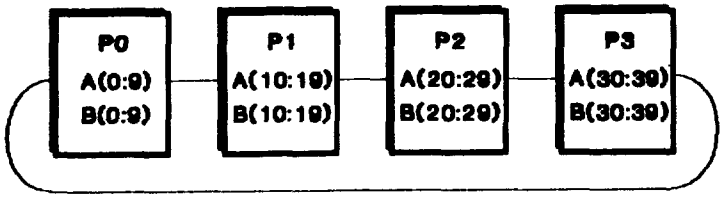


Using a domain decomposition approach, two things need to be specified: the partitioning of the data among the processors and the communication between the processors. Current research assigns the partitioning task to the user (via directives or additional statements [CaK88, KMV87, RoP89]; the task of identifying and inserting communication points is then left to the compiler. We make the same assumption and show how parallel loops can be mapped into distributed code by prefetching all required data before doing any computation. We show how prefetching can introduce broadcast operations for certain types of data accesses in loops (which we call *systolic loops)* which may not be desirable. To remove the broadcast, we use variants of the wavefront method, at a loss of efficiency.

We then introduce a new program transformation specifically targeted for mapping systolic loops onto DMMPs. The new program transformation, called *loop rotation*, is described as another variant of the two-dimensional wavefront method, but designed to improve interprocessor communication characteristics of the loop rather than to uncover more parallelism. When it applies, loop rotation produces an efficient program with uniform communication path utilization. We then discuss how to apply loop rotation in a three-dimensional loop, where it improves efficiency but does not produce a perfectly parallel algorithm. Finally, we use loop rotation as a mechanism to compile code for which the data was distributed in a block-diagonal or block-anti-diagonal manner.

2. Motivation

This section considers how shared-memory parallel loops can be mapped onto DMMPs. Consider a ring-connected DMMP, with data distributed evenly among the processors, as below:



If A and B are matrices, we assume that only one of the dimensions will be distributed, so each processor will own ⌈N/NUMP⌉ columns (or rows). We emphasize that the distribution is given to the system by the user. A simple parallel loop with no communication, such as:

Program 1:

```
doall I = 0:N-1
  A(I) = A(I) + B(I)
enddo
```

could be compiled into the parallel code:

Program 2:

```
on each processor P = 0:NUMP-1
  NN = CEIL(N/NUMP)
  PL = P*NN
  PU = MIN(N-1, (P+1)*NN-1)
  do I = PL, PU
    A(I) = A(I) + B(I)
  enddo
```

where the P index specifies parallel execution on all processors. We assume that the user does not specify or even know the actual number of processors on which his code will be running, so he will not know how much of each array will be stored on any processor. In cases where there is communication between iterations, the compiler must insert communication; several methods are shown below.

2.1. Prefetching

In a simple loop, such as:

Program 3:

```
doall I = 1:N-2
  A(I) = 0.5*(B(I-1) + B(I+1))
enddo
```

the compiler would realize that some of the B(I-1) and B(I+1) references would be to elements stored in neighboring processors, and could prefetch those elements before executing the body of the loop:

Program 4:

```
on each processor P = 0:NUMP-1
  NN = CEIL(N/NUMP)
  PL = MAX(1, P*NN)
  PU = MIN(N-1, (P+1)*NN-1)
  if( P > 0 ) send( B(PL), P-1 )
  if( P < NUMP-1 ) send( B(PU), P+1 )
  if( P < NUMP-1 ) recv( B(PU+1), P+1 )
  if( P > 0 ) recv( B(PL-1), P-1 )
  do I = PL, PU
    A(I) = 0.5*(B(I-1) + B(I+1))
  enddo
```

The more complicated example below requires a more sophisticated approach:

Program 5:

```
doall I = 0:N-1
  do J = 0, N-1
    A(I) = A(I) + B(J)
  enddo
enddo
```

We can apply the "prefetch" technique described above; since each processor needs the whole vector B, each processor would prefetch the whole vector B into local memory before entering the parallel loop. On a DMMP, this requires the "owner" of each segment of B to execute an explicit broadcast:

Program 6:

```
on each processor P = 0:NUMP-1
  NN = CEIL(N/NUMP)
  PL = P*NN
  PU = MIN(N-1, (P+1)*NN-1)
  do JP = 0, NUMP-1
    JL = JP*NN
    JU = MIN(N-1, (JP+1)*NN-1)
    if( JP = P ) then
      broadcast( B(JL:JU) )
    else
      recv( B(JL:JU), JP )
    endif
  enddo
  do I = PL, PU
    do J = 0, N-1
      A(I) = A(I) + B(J)
    enddo
  enddo
```

Now all references to B in the computation loop are to local memory; however, in addition to the overhead of the broadcasts, each processor must have enough memory to hold the entire vector B. This may seem a trivial concern, but one of the common design characteristics of distributed memory systems is to attach each processor to a relatively small local memory. Also, as mentioned, each element of B shown above may actually be a column of a matrix.

2.2. Interleaving

To get around this problem, the compiler may instead try to interleave the broadcasts with the computation:

Program 7:

```
on each processor P = 0:NUMP-1
  NN = CEIL(N/NUMP)
  PL = P*NN
  PU = MIN(N-1, (P+1)*NN-1)
  do JP = 0, NUMP-1
    JL = JP*NN
    JU = MIN(N-1, (JP+1)*NN-1)
    if( JP = P ) then
      broadcast( B(JL:JU) )
    else
      recv( B(JL:JU), JP )
    endif
    do I = PL, PU
      do J = JL, JU
        A(I) = A(I) + B(J)
      enddo
    enddo
  enddo
```

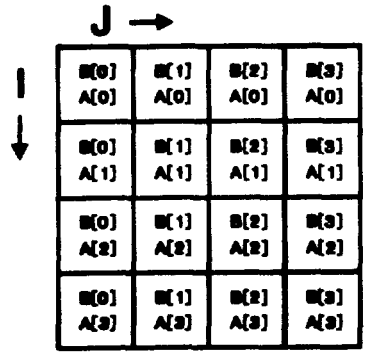
Now each processor only needs enough local memory to store one extra "block" of B in addition to its own; for simplicity, this program does not show the extra storage that would be required for the extra block of B. The broadcasts constitute overhead that can be overlapped with the computation if asynchronous communication is allowed. Each iteration through the loop can initiate the broadcast and receive that will be needed in the subsequent iteration of the loop; the communication can take place while the computation for this iteration is going on. The generated code is messy, so a simplified version is shown below:

Program 8:

```
on each processor P = 0:NUMP-1
  NN = CEIL(N/NUMP)
  PL = P*NN
  PU = MIN(N-1, (P+1)*NN-1)
  if( P = 0 ) then
    abroadcast( B(PL:PU) )
  else
    arecv( B(PL:PU), 0 )
  endif
  do JP = 0, NUMP-1
    JL = JP*NN
    JU = MIN(N-1, (JP+1)*NN-1)
    NJL = (JP+1)*NN + 1
    NJU = MIN(N, (JP+2)*NN)
    if( JP+1 = P ) then
      abroadcast( B(NJL:NJU) )
    else if( JP < NPROC )
      arecv( B(NJL:NJU), JP+1 )
    endif
    do I = PL, PU
      do J = JL, JU
        A(I) = A(I) + B(J)
      enddo
    enddo
  enddo
```

where **abroadcast** and **arecv** refer to asynchronous communication; we have not shown the code that would be required to check that the data was actually received before it was used, or the double buffering that would be used to prevent an **arecv** from overwriting the data buffer too early. This method would be satisfactory except for the broadcasts; broadcasts may be too expensive to amortize efficiently over the computation.

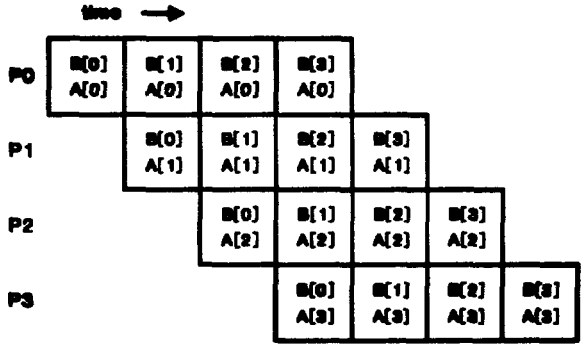
We can look at the iteration space of Program 5 as partitioned by the methods above; both the I and J dimensions of the iteration space are partitioned due to the accesses of A (aligned with the I dimension) and B (aligned with J).



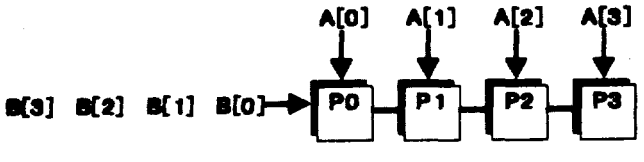
We consider execution of each partition (or tile) to be atomic, so a time step is the time to execute a whole partition. The methods of programs 6-8 assign processors along the I axis with time flowing along the J axis. The problem is that vertically aligned blocks of the iteration space, which will be executed in parallel, use the same partitions of B, requiring broadcasts.

2.3. Wavefronts

We can instead "wavefront" the loop to remove the broadcasts. The wavefront method can be implemented by skewing the iteration space, as shown below, while still assigning processors along the I axis with time flowing to the right:

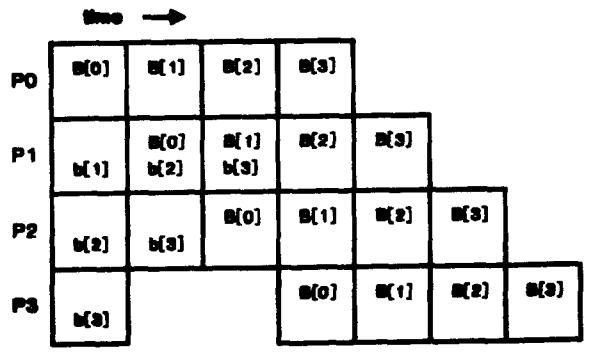


At any time step, each processor uses distinct blocks of the B matrix. In fact, each block of the B matrix is first processed by processor 1, then can be shifted down to processor 2, and so on. This is essentially a "systolic" formulation of the algorithm, with the computation scheduled on the processors when the data arrives. A systolic array, however, would have the A and B vectors arriving from some external environment:



In our system model the data is already distributed among the processors. We can still use a systolic formulation of the algorithm by having the processors 2 through NUMP shift their data up towards processor 1 for the first NUMP-1 time steps. This requires that during some time steps, some processors are shifting data and performing computation during some time steps; the lower case letters in the figure below correspond to blocks of data being shifted without partici-

pating in computation:

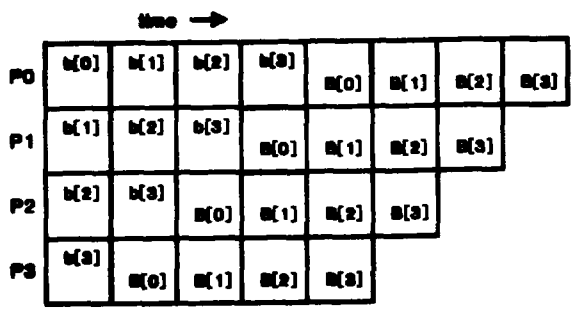


The wavefront formulation solves the problem of broadcasts, but with a loss of efficiency; for large numbers of processors, the efficiency for doubly-nested loops such as this is close to 50%. In addition, data must flow up towards processor 1 (priming the pump), then back down the processor ring, essentially bouncing off processor 1. Some processors have an unbalanced load, having to pass along data from lower processors and compute with and pass along data from upper processors. Some communication paths have twice the usage of others during certain time periods.

The wavefront method is usually used to uncover parallelism in a loop where data dependence constraints prevent parallel execution of either loop [Lam74, Wol86]. In our example this is not the case; in fact, the outer loop of the algorithm is explicitly parallel. However, to map the parallel loop onto the processor ring, we are required to introduce some sequentiality in order to allow the data to reach the processor at the time at which it is needed.

2.4. Reverse Wavefront

To remove the bidirectional data movement problem as well as the unbalanced communication link problem, we can skew the iteration space the other way, letting processor NUMP lead the way. The data flows upward along the processor array:



We still have a "pump-priming" phase, except now it sends data around the ring connection. All data communication flows in the same direction around the ring (uniform data flow) and no communication path or processor has an unbalanced load. The only unsolved problem is the 50% efficiency.

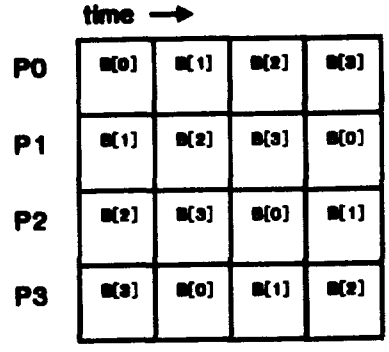
3. Rotation

Using Program 5 as our example, we want to generate a distributed parallel loop with several characteristics:

1) Uniform data communication (as in the reverse wavefront).
2) Balanced communication and computation across processors.

3) High efficiency.

If we again study the iteration space of the loop as partitioned along the I and J axes, we see that each processor has the data necessary to start executing the diagonal blocks. The reverse wavefront formulation aligns these blocks of the iteration space; what we wish to do next is to reorder the execution in each row of the iteration space so each processor starts executing at the diagonal elements:



This is essentially taking the reverse wavefront iteration space and moving the priming phase to fit in with the flushing phase; we call this *loop rotation.*

In the rotated iteration space, each processor can start computing immediately, since it already has all the data for its first block; thus, the efficiency requirement is satisfied. Between blocks (along the horizontal axis), all data communication moves upward and around the ring; this produces uniform and balanced communication. Again, we emphasize that loop rotation does not add parallelism to an algorithm; the parallelism was already there. The problem was the communication constraints, and the desire to remove the broadcasts.

3.1. Requirements

Under what conditions is loop rotation legal? If we consider only the case where the outer loop is already parallel, then the only dependence relations allowed in the iteration space are along the inner axis. If there are no such dependence relations (no dependences *carried* by the inner loop [AlK87]), then rotation is clearly legal. Program 5 does not satisfy this condition, since it accumulates a sum in the inner loop. Reordering the summation will produce the same answer, except for the difference in roundoff error accumulation. Vectorizing compilers typically have a switch which a user can toggle telling the compiler whether the roundoff error differences are acceptable to the user or not; some reduction operations can be reordered without fear of roundoff error accumulation [Wol89]. Usually, for well-conditioned problems, the difference is roundoff error accumulation is acceptable, especially with the gain in performance. We use the same idea here, allowing reordering of associative reductions.

Loop rotation really only applies for nested loops with two characteristics:

1) the loop nesting is greater than the dimensionality of the processor topology;

2) some distributed dimension is accessed by more than one parallel loop.

In Program 5, we have a one-dimension processor ring with a loop nest depth of two, and the distributed dimension of A is accessed by the I loop while the corresponding distributed dimension of B is accessed by the J loop. Another example of this would be a matrix multiply; suppose we have the same one-dimensional ring of processors, and three matrices, A, B and C, all distributed by columns (second dimension):

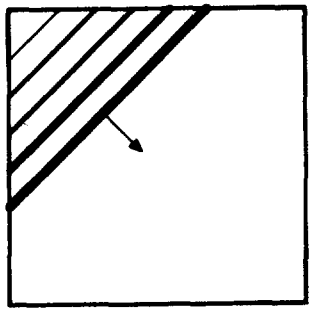
Program 9:

```
doall I = 0, N-1
  doall J = 0, N-1
    do K = 0, N-1
      A(I,J) = A(I,J) + B(I,K)*C(K,J)
    enddo
  enddo
enddo
```

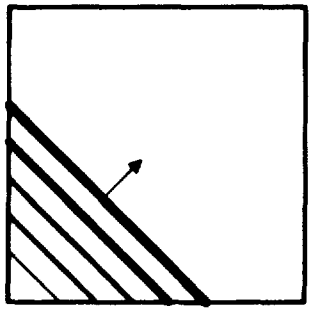
In this case, we ignore the I loop, since it indexes row number, and all rows of a particular column will be stored on a single processor. The pertinent loops are J and K; we then have two nested loops on a one-dimensional ring of processors, with the columns of A and C accessed by J, and columns of B accessed by K. This leads us to believe that loop rotation may be of some value here. In fact, if we ignore the first dimension and the I loop, we have a program very like Program 5, and it would be handled the same way.

3.2. Rotation vs. Wavefronts

The wavefront method of executing a loop can be described as moving a slanted line through the iteration space, and executing in parallel all iterations which simultaneously intersect with the slanted line. With a classical wavefront, the slanted line will first intersect with the upper left corner of the iteration space, then move down and to the right:

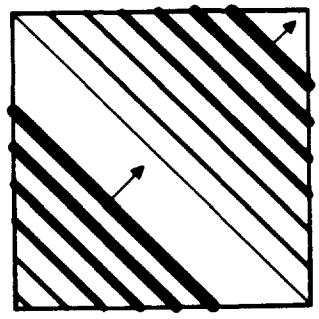


In the reverse wavefront we used to derive loop rotation, the slanted line first intersects the lower left corner of the iteration space, then moves up and to the right:



Loop rotation can be described as treating the iteration space as a cylinder (instead of a flat rectangle), moving a slanted line through the iteration space (wrapping around the cylinder), and executing in parallel all iterations which simultaneously intersect with the line. The slanted line starts out going through the diagonal elements, then moves to the right and around

the iteration space:



At this point we note that the wavefront method of executed a nested loop involves skewing the iteration space, then ordering the skewed loops such that sequential execution of the outermost loop will satisfy all dependence relations and allow parallel execution of all inner loops [Wol86]. In the two dimensional version, the sequential wavefront loop surrounds a single parallel inner loop; in a three dimensional wavefront, the sequential wavefront loop surrounds two nested parallel inner loops; the one-dimensional analog of a wavefront is just a single sequential loop. Loop rotation corresponds to converting a two dimensional wavefront to a vector of one-dimensional wavefronts in the rotated iteration space. We will see why this view is so interesting in three dimensional loop rotation.

Loop skewing is simply defined as changing the shape of the iteration space by adding the outer loop index to the lower and upper limits of the inner loop; this requires that the subtracting the outer loop index from the inner loop index within the body of the loop. Thus program 5, after skewing the inner loop, becomes:

Program 10:

```
doall I = 0:N-1
  do J = I+0, I+N-1
    A(I) = A(I) + B(J-I)
  enddo
enddo
```

A classical wavefront is derived by interchanging these two loops [Wol86]. A reverse wavefront will subtract the outer loop index from the inner loop limits:

Program 11:

```
doall I = 0:N-1
  do J = 0-I, N-I-1
    A(I) = A(I) + B(J+I)
  enddo
enddo
```

A loop rotation optimization is defined simply when the loop lower limits are zero; the inner loop is 'rotated' by adding the outer loop index to the inner loop index within the body of the loop, modulo the trip count of the loop:

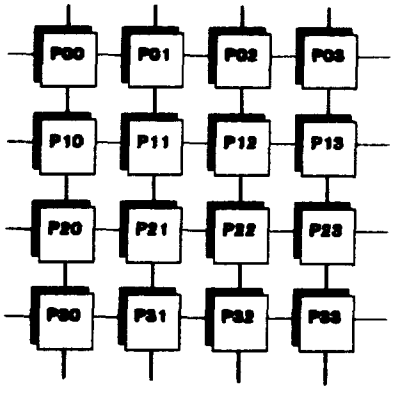
Program 12:

```
doall I = 0:N-1
  do J = 0, N-1
    A(I) = A(I) + B((J+I) mod N)
  enddo
enddo
```

The way we have defined the optimizations, we would first partition (or tile) the iteration space according to the data distribution, and we would skew or rotate only the tile loops.

4. Loop Rotation in 3D

Now consider a two-dimensional mesh (torus, actually) of processors, with matrices distributed evenly across the ensemble:



Simple two-nested loops could be handled by prefetching data into the processors before performing computation (as with the single-nested loops on a one-dimensional ring of processors). Here we concentrate on loops nested three deep, where some distributed dimension of the matrices is indexed by more than one loop variable. One example would be where a single dimension is indexed by two loop variables:

Program 13:

```
doall I = 0, N-1
  doall J = 0, N-1
    do K = 0, N-1
      A(I,J) = A(I,J) + B(I,K)
    enddo
  enddo
enddo
```

It is easy to see that this case can be handled the same way as the two dimensional case in the previous section. The original iteration space (for N=4) is shown in figure 1; assigning processors to the I×J axes (with time flowing along the K axis) requires that all processors with the same I address need the same block of B at the same time. Using a reverse wavefront in J and K gives the modified iteration space in figure 2. Assigning processors to the I×J axes has the properties we want; again, time flows to the right, so vertically aligned blocks in the iteration space are executing in parallel. No two vertically aligned blocks require the same data; moreover, only nearest neighbor connections are required. The A matrix stays in the same processor, while B moves around the processors backwards (upwards) in the J dimension. Figure 3 shows the rotated iteration space (K rotated around J), which has the property that all processors can simultaneously with all local data, and the only data motion is blocks of B moving around the J dimension. This program essentially a vector of two-dimensional rotated loops, and so is nothing new.

A more interesting problem arises in a triply nested loop where both dimensions are indexed by more than one loop variable. Our old friend matrix multiply (program 9) is a perfect such example. We want to derive a formulation for the program on a toroid DMMP with high efficiency, uniform communication and no broadcasts.

The original partitioned iteration space is shown in figure 4. Simply skewing the iteration space analogous to the 2D case gives the iteration space in figure 5; processors are assigned down the columns (I×J axes) and time runs to the right (K axis). Note that each processor always refers to the same block of the A matrix, the B matrix moves across processors down the J axis and the C matrix moves across processors down the I axis. As in the 2D case, this corresponds to a systolic algorithm. This wavefront formulation requires the blocks of B data to be moved up to the (*,0) processors and the blocks of C to be moved up to the (0,*) processors; while no broadcasts are required, data is moving in two directions (upwards to prime the

pump, and downwards for the systolic computation). Again, as in the 2D case, we solve the nonuniform communication problems by using a reverse wavefront; the iteration space for the 3D reverse wavefront is shown in figure 6. Here data moves uniformly up each dimension all the time, though it is not involved in any computation until it reaches the last processor in that dimension. We still have low efficiency, however.

We are tempted to attack the low efficiency by finding a 3D loop rotation transformation, since the 2D loop rotation generated perfect efficiency. What would be the characteristics of a perfectly efficient rotation transformation? We need to find $\sqrt{\text{NUMP}} \times \sqrt{\text{NUMP}}$ blocks in the iteration space where, for each block, all the data needed to start that block is initially resident in a single processor; this allows us to start all these processors all at once (assuming they are distinct). Then we need to order the rest of the iterations in such a fashion that data flows naturally between nearest neighbor processors in uniform directions.

The news on these points is not good. Let us first examine the efficiency question, since the reverse wavefront already has uniform communication. We derived 2D loop rotation from 2D reverse wavefront by noticing that halfway through, all the diagonal blocks were aligned and all were using data that was initially stored on those processors; by starting the processors at the diagonal, loop rotation fell out. In 3D, again only the diagonal blocks use data which is initially stored on the processor executing those blocks; unfortunately, there are only $\sqrt{\text{NUMP}}$ diagonal blocks. Moreover, when we look at the reverse wavefront iteration space (figure 6), those diagonal blocks don't even line up; `(0,0,0)` is executed before `(1,1,1)`, and so on.

What might happen if we did initiate those blocks? Not surprisingly, the 3D analog of loop rotation is a vector of 2D wavefronts; the 3D rotated iteration space is shown in figure 7. Figure 8 shows the original iteration space with the time steps at which each block in the iteration space would be executed; each 2D wavefront starts at a diagonal iteration and spreads out to the right and down within a plane of the 3D iteration space. There are three possible orientations of the vector of wavefronts, so we choose the one that allows the result matrix (A) to reside on the same processor throughout the loop. Thus, loop rotation in general seems to replace a *d* dimensional wavefront by a vector of *(d-1)* dimensional wavefronts. In the 2D case, the result is perfect efficiency; for higher dimensionality, the savings is not so great and may not be worth the trouble if it brings added complexity.

Now for the added complexity. Looking carefully at figure 7, we see the data blocks required by each processor at each time step. The orientation of the rotation was chosen to hold the A matrix blocks fixed within processors, while the communication pattern for the B matrix blocks is very regular. The pattern for C is very irregular. Notice that in the third time step, `C(2,0)` and `C(0,2)` are each used by two different processors simultaneously; the next time step uses each `C(2,1)` and `C(1,2)` simultaneously in two processors. While duplication is not necessarily infeasible, the communication pattern is very irregular. Transposing the matrix or other simple fixes will not help matters. It seems that the complexity required by 3D loop rotation may well prevent its use in the only programs that could benefit from it.

5. Diagonal Distributions

Given a ring of processors, there are distribution mechanisms other than by rows or by columns; the one shown here was first described to the author by Martin Shultz of Yale Univ. Suppose we divide each matrix into NUMP×NUMP square blocks, and assign block `(I,J)` to processor `I-J mod NUMP` (to get diagonal distribution, shown in figure 9) or `I+J mod NUMP` (to get anti-diagonal distribution, shown in figure 10). These distributions have the advantage that both array access by row and by column can be done in parallel.

With parallel loops using this distribution mechanism, we want a method to identify the proper iterations to execute on each processor. Loop rotation will adjust the iteration space to align either the diagonal or anti-diagonal down one dimension of the iteration space, allowing parallel execution across the other dimension.

6. Epilogue

To map parallel loops in shared-memory programs onto distributed memory message passing multiprocessors with predefined data partitioning, we must identify the data that needs to be communicated between processors. In what we call systolic loops, prefetching all the data will be too expensive, in time and local storage. The best loop schedule would have several key characteristics:

1) good processor efficiency (all processors busy doing useful work)

2) uniform communication patterns

3) no broadcasts

We showed how loop skewing or wavefronting can remove broadcasts, and how reverse skewing or reverse wavefronts can provide uniform communication patterns. To address processor efficiency we introduce a new program transformation, called loop rotation, which changes the order of execution of the iterations in the loop. This may not always be feasible, but it has certain advantages when it is. In the 2D case (two dimensional loops on a one dimensional ring of processors), loop rotation (when it applies) can generate perfect efficiency while still satisfying the other desires. In the 3D case, however, loop rotation falls apart; the 3D rotation corresponds to changing from a 3D wavefront to a vector of 2D wavefronts. The efficiency of a 3D wavefront is (roughly) 33%, while the efficiency of a 2D wavefront is 50%; thus, rotation will not even double the efficiency. The 3D rotation does not have uniform communication patterns for all data, and some data is even required in multiple places at the same time.

While it is not clear that loop rotation scales beyond the 2D case at all, it is nonetheless interesting within its limited field of application. We plan to continue study of rotation and other such program transformations for distributed memory computers. It is worthwhile to emphasize that loop rotation was not used to uncover any latent parallelism in the algorithm; in fact, our examples all used explicit parallelism. Rotation (and other transformations) will be used to efficiently map parallel algorithms onto distributed machines. This problem has some relation to the systolic algorithm mapping problem, but has the additional constraints of a fixed data partition among the processors.

References

References

[AlK87] J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987), 491-542 .

[CaK88] D. Callahan and K. Kennedy, Compiling Programs for Distributed-memory Multiprocessors, *The Journal of Supercomputing 2*, 2 (August 1988), .

[KMV87] C. Koelbel, P. Mehrotra and J. Van Rosendale, Semi-automatic Process Partitioning for Parallel Computation, *International J. of Parallel Programming 16*, 5 (October 1987), 365-382.

[Lam74] L. Lamport, The Parallel Execution of DO Loops, *Communications of the ACM 17*, 2 (February 1974), 83-93, ACM.

[RoP89] A. Rogers and K. Pingali, Process Decomposition Through Locality of Reference, in *to appear in ACM SIGPLAN Notices '89 Conf. on Programming Language Design and Implementation*, 1989.

[Wol86] M. Wolfe, Loop Skewing: The Wavefront Method Revisited, *Int'l Journal of Parallel Programming 15*, 4 (August 1986), 279-294.

[Wol89] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing and MIT Press, London and Boston, 1989.

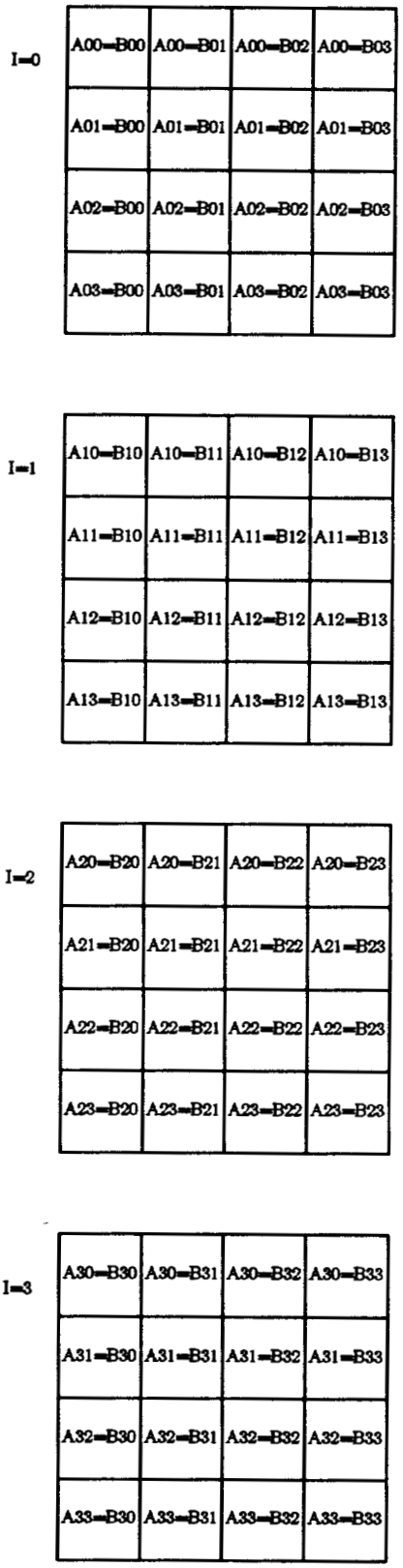

I=0
A00=B00 A00=B01 A00=B02 A00=B03
A01=B00 A01=B01 A01=B02 A01=B03
A02=B00 A02=B01 A02=B02 A02=B03
A03=B00 A03=B01 A03=B02 A03=B03
I=1
A10=B10 A10=B11 A10=B12 A10=B13
A11=B10 A11=B11 A11=B12 A11=B13
A12=B10 A12=B11 A12=B12 A12=B13
A13=B10 A13=B11 A13=B12 A13=B13
I=2
A20=B20 A20=B21 A20=B22 A20=B23
A21=B20 A21=B21 A21=B22 A21=B23
A22=B20 A22=B21 A22=B22 A22=B23
A23=B20 A23=B21 A23=B22 A23=B23
I=3
A30=B30 A30=B31 A30=B32 A30=B33
A31=B30 A31=B31 A31=B32 A31=B33
A32=B30 A32=B31 A32=B32 A32=B33
A33=B30 A33=B31 A33=B32 A33=B33

Figure 1.

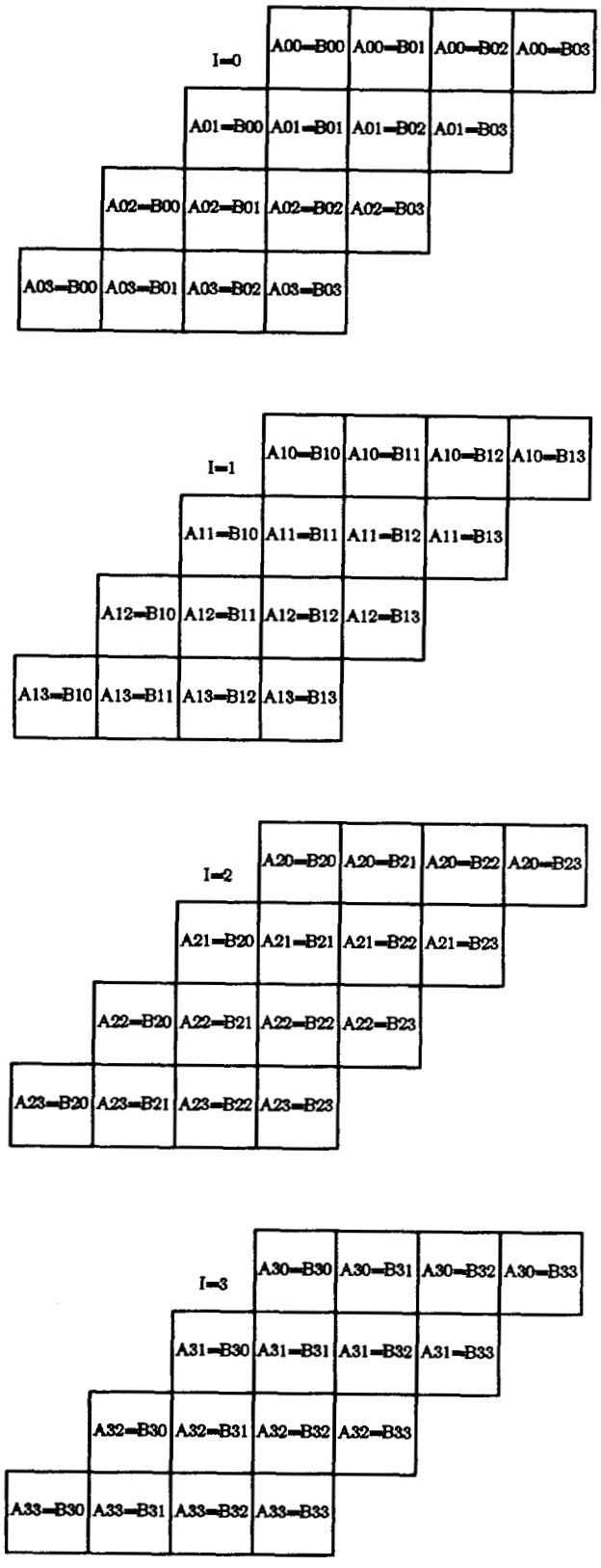

I=0
A00=B00 A00=B01 A00=B02 A00=B03
A01=B00 A01=B01 A01=B02 A01=B03
A02=B00 A02=B01 A02=B02 A02=B03
A03=B00 A03=B01 A03=B02 A03=B03
I=1
A10=B10 A10=B11 A10=B12 A10=B13
A11=B10 A11=B11 A11=B12 A11=B13
A12=B10 A12=B11 A12=B12 A12=B13
A13=B10 A13=B11 A13=B12 A13=B13
I=2
A20=B20 A20=B21 A20=B22 A20=B23
A21=B20 A21=B21 A21=B22 A21=B23
A22=B20 A22=B21 A22=B22 A22=B23
A23=B20 A23=B21 A23=B22 A23=B23
I=3
A30=B30 A30=B31 A30=B32 A30=B33
A31=B30 A31=B31 A31=B32 A31=B33
A32=B30 A32=B31 A32=B32 A32=B33
A33=B30 A33=B31 A33=B32 A33=B33

Figure 2.

I=0

A00=B00	A00=B01	A00=B02	A00=B03
A01=B01	A01=B02	A01=B03	A01=B00
A02=B02	A02=B03	A02=B00	A02=B01
A03=B03	A03=B00	A03=B01	A03=B02

I=1

A10=B10	A10=B11	A10=B12	A10=B13
A11=B11	A11=B12	A11=B13	A11=B10
A12=B12	A12=B13	A12=B10	A12=B11
A13=B13	A13=B10	A13=B11	A13=B12

I=2

A20=B20	A20=B21	A20=B22	A20=B23
A21=B21	A21=B22	A21=B23	A21=B20
A22=B22	A22=B23	A22=B20	A22=B21
A23=B23	A23=B20	A23=B21	A23=B22

I=3

A30=B30	A30=B31	A30=B32	A30=B33
A31=B31	A31=B32	A31=B33	A31=B30
A32=B32	A32=B33	A32=B30	A32=B31
A33=B33	A33=B30	A33=B31	A33=B32

Figure 3.

I=0

A00=B00 C00	A00=B01 C10	A00=B02 C20	A00=B03 C30
A01=B00 C01	A01=B01 C11	A01=B02 C21	A01=B03 C31
A02=B00 C02	A02=B01 C12	A02=B02 C22	A02=B03 C32
A03=B00 C03	A03=B01 C13	A03=B02 C23	A03=B03 C33

I=1

A10=B10 C00	A10=B11 C10	A10=B12 C20	A10=B13 C30
A11=B10 C01	A11=B11 C11	A11=B12 C21	A11=B13 C31
A12=B10 C02	A12=B11 C12	A12=B12 C22	A12=B13 C32
A13=B10 C03	A13=B11 C13	A13=B12 C23	A13=B13 C33

I=2

A20=B20 C00	A20=B21 C10	A20=B22 C20	A20=B23 C30
A21=B20 C01	A21=B21 C11	A21=B22 C21	A21=B23 C31
A22=B20 C02	A22=B21 C12	A22=B22 C22	A22=B23 C32
A23=B20 C03	A23=B21 C13	A23=B22 C23	A23=B23 C33

I=3

A30=B30 C00	A30=B31 C10	A30=B32 C20	A30=B33 C30
A31=B30 C01	A31=B31 C11	A31=B32 C21	A31=B33 C31
A32=B30 C02	A32=B31 C12	A32=B32 C22	A32=B33 C32
A33=B30 C03	A33=B31 C13	A33=B32 C23	A33=B33 C33

Figure 4.

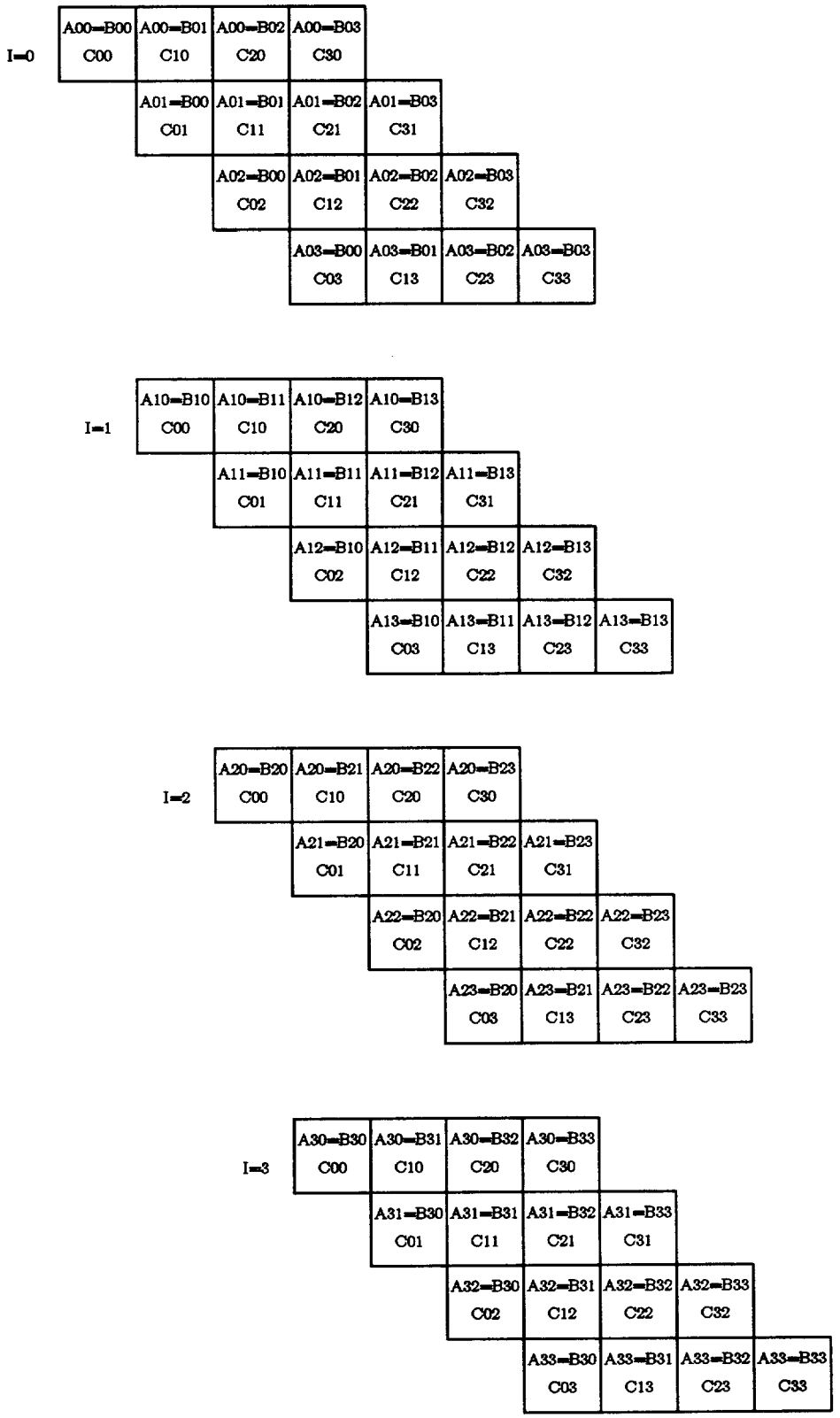

I=0
A00=B00 C00
A00=B01 C10
A00=B02 C20
A00=B03 C30
A01=B00 C01
A01=B01 C11
A01=B02 C21
A01=B03 C31
A02=B00 C02
A02=B01 C12
A02=B02 C22
A02=B03 C32
A03=B00 C03
A03=B01 C13
A03=B02 C23
A03=B03 C33
I=1
A10=B10 C00
A10=B11 C10
A10=B12 C20
A10=B13 C30
A11=B10 C01
A11=B11 C11
A11=B12 C21
A11=B13 C31
A12=B10 C02
A12=B11 C12
A12=B12 C22
A12=B13 C32
A13=B10 C03
A13=B11 C13
A13=B12 C23
A13=B13 C33
I=2
A20=B20 C00
A20=B21 C10
A20=B22 C20
A20=B23 C30
A21=B20 C01
A21=B21 C11
A21=B22 C21
A21=B23 C31
A22=B20 C02
A22=B21 C12
A22=B22 C22
A22=B23 C32
A23=B20 C03
A23=B21 C13
A23=B22 C23
A23=B23 C33
I=3
A30=B30 C00
A30=B31 C10
A30=B32 C20
A30=B33 C30
A31=B30 C01
A31=B31 C11
A31=B32 C21
A31=B33 C31
A32=B30 C02
A32=B31 C12
A32=B32 C22
A32=B33 C32
A33=B30 C03
A33=B31 C13
A33=B32 C23
A33=B33 C33

Figure 5.

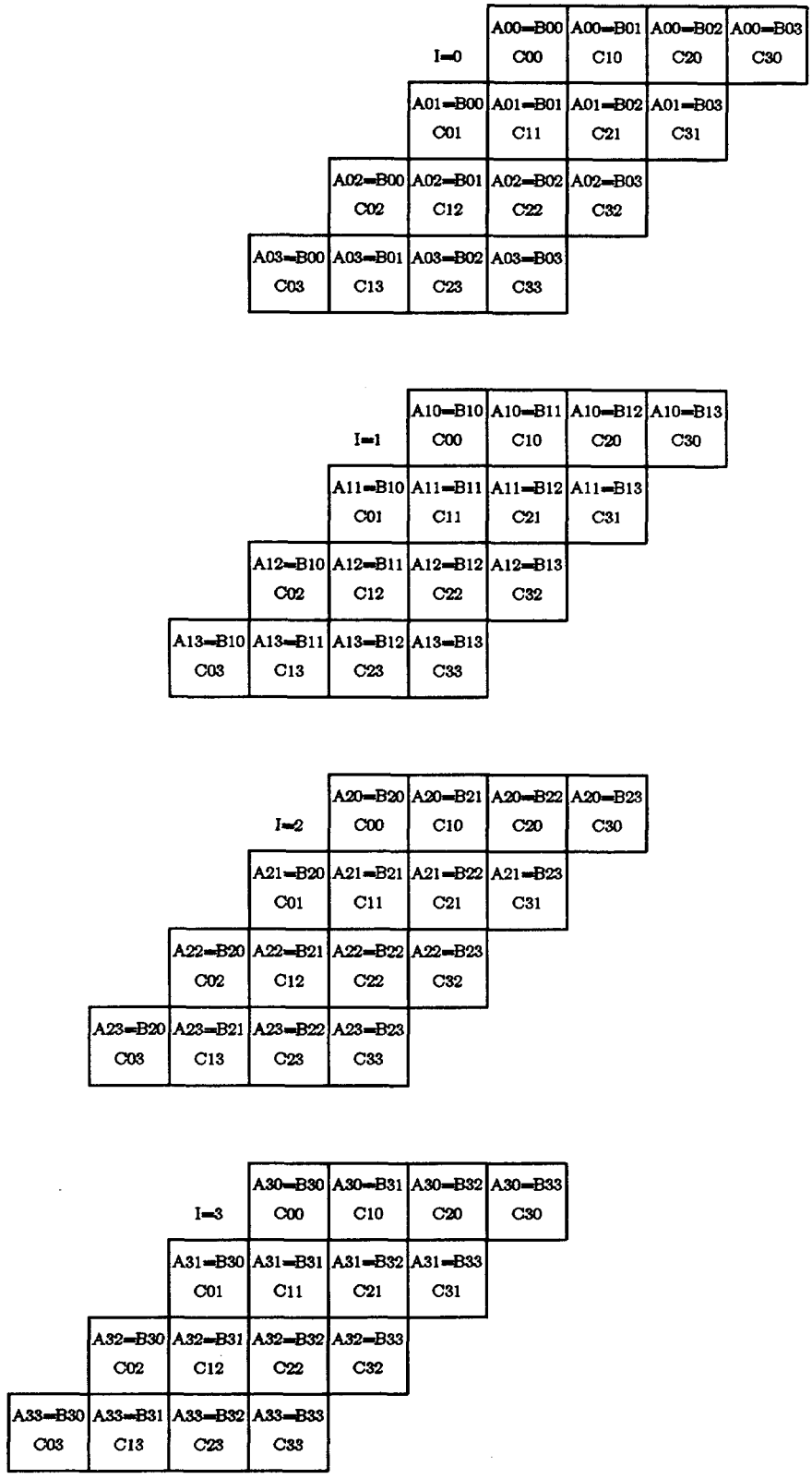

I=0
A00=B00 C00
A00=B01 C10
A00=B02 C20
A00=B03 C30
A01=B00 C01
A01=B01 C11
A01=B02 C21
A01=B03 C31
A02=B00 C02
A02=B01 C12
A02=B02 C22
A02=B03 C32
A03=B00 C03
A03=B01 C13
A03=B02 C23
A03=B03 C33
I=1
A10=B10 C00
A10=B11 C10
A10=B12 C20
A10=B13 C30
A11=B10 C01
A11=B11 C11
A11=B12 C21
A11=B13 C31
A12=B10 C02
A12=B11 C12
A12=B12 C22
A12=B13 C32
A13=B10 C03
A13=B11 C13
A13=B12 C23
A13=B13 C33
I=2
A20=B20 C00
A20=B21 C10
A20=B22 C20
A20=B23 C30
A21=B20 C01
A21=B21 C11
A21=B22 C21
A21=B23 C31
A22=B20 C02
A22=B21 C12
A22=B22 C22
A22=B23 C32
A23=B20 C03
A23=B21 C13
A23=B22 C23
A23=B23 C33
I=3
A30=B30 C00
A30=B31 C10
A30=B32 C20
A30=B33 C30
A31=B30 C01
A31=B31 C11
A31=B32 C21
A31=B33 C31
A32=B30 C02
A32=B31 C12
A32=B32 C22
A32=B33 C32
A33=B30 C03
A33=B31 C13
A33=B32 C23
A33=B33 C33

Figure 6.

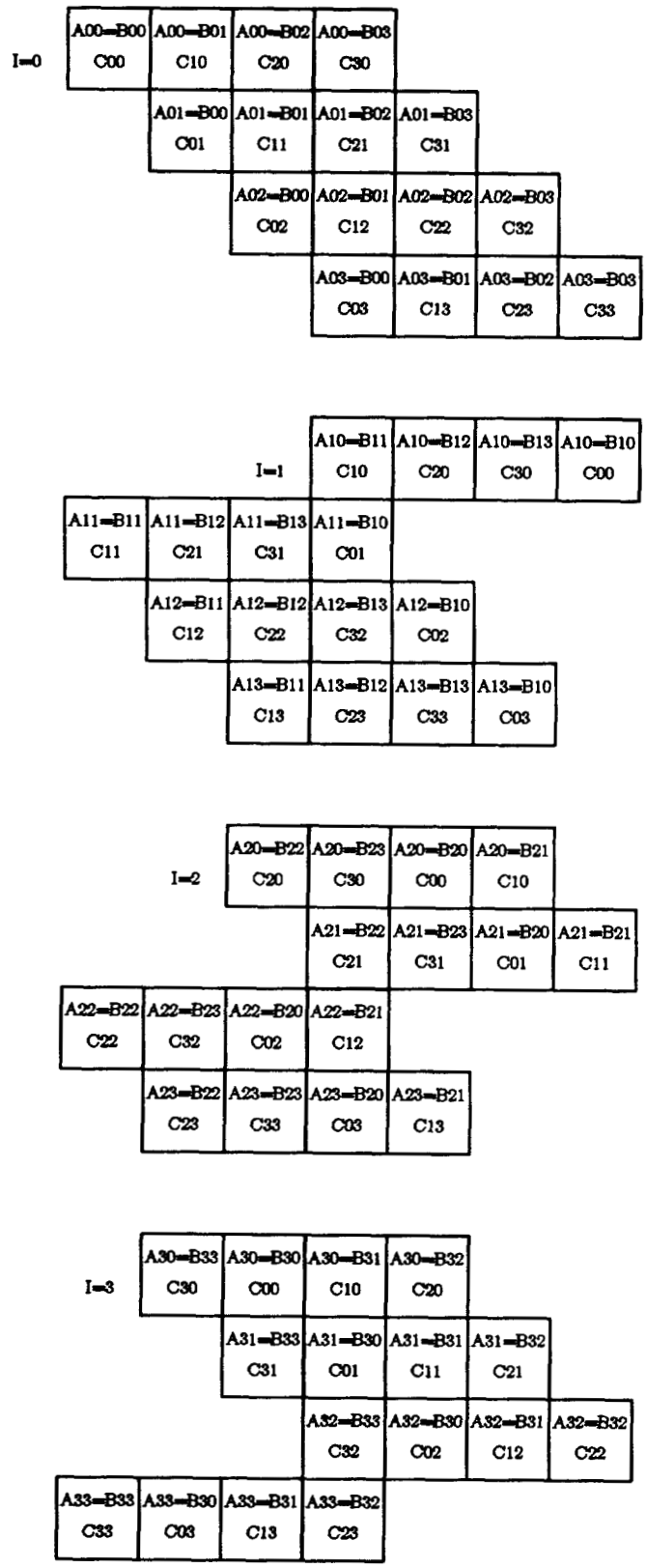

I=0
A00=B00 C00
A00=B01 C10
A00=B02 C20
A00=B03 C30
A01=B00 C01
A01=B01 C11
A01=B02 C21
A01=B03 C31
A02=B00 C02
A02=B01 C12
A02=B02 C22
A02=B03 C32
A03=B00 C03
A03=B01 C13
A03=B02 C23
A03=B03 C33
I=1
A10=B11 C10
A10=B12 C20
A10=B13 C30
A10=B10 C00
A11=B11 C11
A11=B12 C21
A11=B13 C31
A11=B10 C01
A12=B11 C12
A12=B12 C22
A12=B13 C32
A12=B10 C02
A13=B11 C13
A13=B12 C23
A13=B13 C33
A13=B10 C03
I=2
A20=B22 C20
A20=B23 C30
A20=B20 C00
A20=B21 C10
A21=B22 C21
A21=B23 C31
A21=B20 C01
A21=B21 C11
A22=B22 C22
A22=B23 C32
A22=B20 C02
A22=B21 C12
A23=B22 C23
A23=B23 C33
A23=B20 C03
A23=B21 C13
I=3
A30=B33 C30
A30=B30 C00
A30=B31 C10
A30=B32 C20
A31=B33 C31
A31=B30 C01
A31=B31 C11
A31=B32 C21
A32=B33 C32
A32=B30 C02
A32=B31 C12
A32=B32 C22
A33=B33 C33
A33=B30 C03
A33=B31 C13
A33=B32 C23

Figure 7.

I=0

T0	T1	T2	T3
T1	T2	T3	T4
T2	T3	T4	T5
T3	T4	T5	T6

I=1

T6	T3	T4	T5
T3	T0	T1	T2
T4	T1	T2	T3
T5	T2	T3	T4

I=2

T4	T5	T2	T3
T5	T6	T3	T4
T2	T3	T0	T1
T3	T4	T1	T2

I=3

T2	T3	T4	T1
T3	T4	T5	T2
T4	T5	T6	T3
T1	T2	T3	T0

Figure 8.

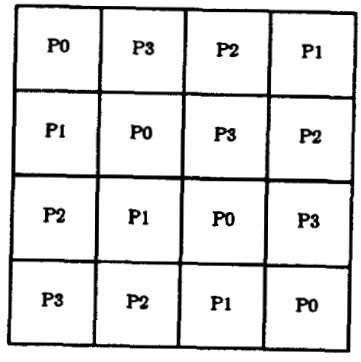


Figure 9.

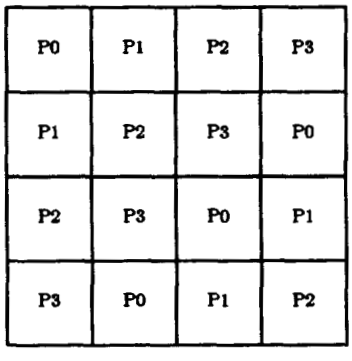


Figure 10.