Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: one	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A Non-systolic Matrix Product Algorithm	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER: 85-11-02	
7. AUTHOR(s): Philip A. Nelson	8. CONTRACT OR GRANT NUMBER(s): N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Washington, Department of Computer Science, Seattle, Washington 98195	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Arlington, VA 22217	12. REPORT DATE: November 1985
	13. NUMBER OF PAGES: 25
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Non-systolic matrix product algorithms: Poker parallel programming environments: XX parallel language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Most parallel matrix-matrix product algorithms for MIMD architectures are systolic. These algorithms can be adapted to be used on a general purpose architecture, such as the CHiP or cube machines. When the processors already contain the matrices, the algorithm can still be used by modifying it to circulate the data as if it was being fed in from an external source. We present a non-systolic matrix product algorithm in which the data movement is not the circulation pattern of the adapted systolic algorithms. Rather, it uses technique similar to Strassen's algorithm. The running time is O(n) using

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entere

A Non-systolic Matrix Product Algorithm

Philip A. Nelson
Computer Science Department, FR35
University of Washington
Seattle, Washington 98195

Technical Report No. 85-11-02
November 1985

Abstract

Most parallel matrix-matrix product algorithms for MIMD architectures are systolic. These algorithms can be adapted to be used on a general purpose architecture, such as the CHiP or cube machines. When the processors already contain the matrices, the algorithm can still be used by modifying it to circulate the data as if it was being fed in from an external source. We present a non-systolic matrix product algorithm in which the data movement is not the circulation pattern of the adapted systolic algorithms. Rather, it uses technique similar to Strassen's algorithm. The running time is O(n) using n^2 processors for n×n matrices. We compare this algorithm to a systolic algorithm and give experimental results.

Supported in part by National Science Foundation Grant DCR-8416878
and by Office of Naval Research Contract No. N00014-85-K-0328.

1. Introduction

There are several algorithms for dense matrix-matrix product on highly-parallel architectures[1-5]. These algorithms are designed for systolic architectures. When using a general purpose highly-parallel machine like the CHiP[6], the systolic algorithm is directly implemented by configuring the CHiP to match the systolic machine. In computations that use matrix product as one operation in a long series of operations, the matrices to be multiplied are likely to be contained within the processors. To use the systolic algorithms, the matrices must be circulated as if they were being fed in from an external source. This provides an easy solution, but perhaps not the most efficient one. In addition, some of these algorithms[2,3,5] require the matrices to be reshaped before the circulation can begin.

This paper presents an algorithm for dense matrix-matrix product that assumes that the data is already contained within the processors. The data does not follow the circulation pattern of the systolic algorithms and no data reshaping is required. Also, this algorithm illustrates the use of the divide-and-conquer paradigm in parallel algorithm design. This algorithm has the advantage that it can be implemented using n^2 or n^3 processors giving running times of O(n) and O(log n), respectively.

In Section 2 the algorithm is presented. An analysis of it's running time is done for n^2 processors. Section 3 describes an implementation of the algorithm using the Poker system[5]. Section 4 describes a matrix product algorithm for the the Wavefront Array Processor(WAP)[4], and its implementation using the Poker system[5]. Section 5 describes a modified form of the WAP algorithm and its implementation. A comparison of the run times of these algorithms for several sizes of matrices is given.

2. The Algorithm

Consider the product of two dense n×n matrices, AB = C, using n^2 processors. Assume that $n = 2^k$ for some constant k. Picture the processors as an n×n array where the processors are labeled PE_{ij} for $1 \le i,j \le n$. The matrices A and B are initially distributed in the n^2 processors such that a_{ij} and b_{ij} are contained in PE_{ij}. After the product we want c_{ij} to be contained in PE_{ij}.

To begin with, consider the 2×2 case. PE_{11} contains a_{11} and b_{11}. To compute c_{11} the values a_{12} and b_{21} are needed. Similarly, all other processors need only 2 elements not already stored at that processor. To provide for direct communication, a grid interconnection structure is used. The processors then send their

a_{ij} value to the other processor in the same row, and their b_{ij} value to the other processor in the same column (see Figure 1). After this communication, each processor, PE_{ij}, has all the data required to compute c_{ij}.

Now consider the n×n case. We use Strassen's[7] matrix decomposition where two n×n matrices can be viewed as two 2×2 matrices of $\frac{n}{2}\times\frac{n}{2}$ matrices. The 2×2 matrices are then multiplied using matrix product and matrix addition on $\frac{n}{2}\times\frac{n}{2}$ matrices.

Let A_{11} be the upper left $\frac{n}{2}\times\frac{n}{2}$ submatrix of A. Also, let the other 3 submatrices be A_{12}, A_{21}, and A_{22}. In the same way, let B_{ij} be the submatrices of B, C_{ij} the submatrices of C, and P_{ij} the subarray of the processors. Then A_{ij} and B_{ij} are contained in P_{ij}. As in the 2×2 case, A_{12} and B_{21} are required to compute C_{11}. If the corresponding processors in P_{11} and P_{12} are directly connected (see Figure 2), A_{12} can be sent to P_{11} in parallel with one communication step. B_{21} can be sent to P_{11} using a similar connection scheme in one communication step. The full connection structure connects PE_{ij} with both $PE_{i\pm\frac{n}{2}j}$ and $PE_{ij\pm\frac{n}{2}}$. With A_{11}, B_{11}, A_{12}, and B_{12} in P_{11}, C_{11} can be computed by doing two $\frac{n}{2}\times\frac{n}{2}$ matrix products and one matrix addition. These products can be done using this same algorithm on the $\frac{n}{2}\times\frac{n}{2}$ matrices. The recursion will stop after k-1 levels when a 2×2 matrix product is done. The matrix addition is performed element by element.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}$$

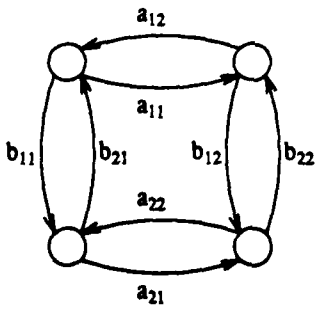


Figure 1: 2×2 product and communication

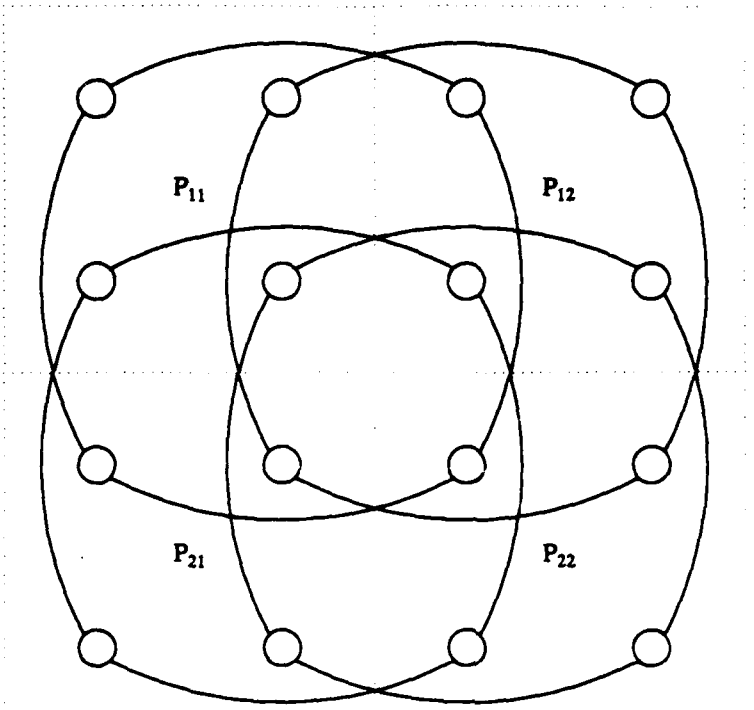


FIGURE 2: 4×4 connections

Each recursion level requires it's own interconnection structure. The complete interconnection structure, supplying an edge for every communication in the algorithm on every level of recursion, is the hypercube.

The time required for this algorithm is O(n). To prove this claim, the recurrence relation for the time is

$$t(n) = 2t_c+t_a+t_o+2t(\frac{n}{2})$$

where $t(2) = 2t_c+t_a+2t_m$, t_c is the time for a communication step, t_a is the time for a scalar addition, t_o is the overhead time for each recursion level and t_m is the time for a scalar multiplication. The closed form is

$$t(n) = (2n-2)t_c+(n-1)t_a+(n-2)t_o+nt_m.$$

To achieve the O(log n) running time, we need to evaluate both $\frac{n}{2}\times\frac{n}{2}$ matrix products at the same time. The algorithm starts with n^2 processors active. These processors contain the original matrices, A and

B. The processors communicate as in the O(n) algorim. At this point, each $\frac{n}{2}\times\frac{n}{2}$ block of processors has two matrix products to compute. One of the products is sent to a set of previously inactive processors. This doubles the number of active processors. The same algorithm is used to compute the "new" products. After the $\frac{n}{2}\times\frac{n}{2}$ products have been computed, the processors that were sent the second product, send back their result. The results of the two products are added element by element to form the result of the n×n matrix product. The recursion for this algorithm stops when a 2×2 product is to be computed. Each processor does both multiplications and the one addition.

To prove the claim of O(log n) time, the recurrence relation is

$$t(n) = 5t_c+t_a+t_o+t(\frac{n}{2})$$

where $t(2)=3t_c+t_a+t_m$, and the constants measure the same quantities as before, but for this algorithm. This recurrence measures the time for the original n^2 active processors. The $5t_c$ comes from two t_c's for the original communication, two t_c's from sending one subproblem to a "new" processor and a t_c for getting the result back from the "new" processor. The closed form is

$$t(n) = (5(\log n\ -1)+3)t_c+\log n\ t_a+(\log n\ -1)t_o+2t_m.$$

This algorithm uses $\frac{n^3}{2}$ processors. It starts with n^2 active processors. After the initial communication, the n^2 processors are divided up into 4, $\frac{n}{2}\times\frac{n}{2}$ sections, each having two matrix products to compute. Every processor sends two values, its part of one matrix product, to an inactive processor, thus activating it. This doubles the number of processors. We now have 8, $\frac{n}{2}\times\frac{n}{2}$ problems using $2n^2$ processors. Each matrix product is then computed by a "recursive call". This is one recursion level. At each successive recursive level the number of active processors is doubled. There are log n −1 levels of recursion. This gives $n^{2^{\log n-1}}$ or $\frac{n^3}{2}$ active processors at the evaluation of the 2×2 products.

3. The Poker Implementation

The Poker system[5] was used to implement this algorithm although Poker and its sequential programming language XX (dos equis) does not directly support recursion. The goals of the implementation

were to follow the recursive algorithm as closely as possible with the nonrecursive system and to reduce the complexity of the communication as much as possible. Recursion was acheived by explicit manipulation of a stack to save data and record the position within the recursive algorithm. The communication was simplified by dividing it into a phase for each level of recursion.

An instance of the algorithm for $n = 2^k$ has $\log n = k$ phases. Each phase has a unique interconnection structure. For reference, we number the phases 1,2,...,k. Phase 1 connects processors together that are $\frac{n}{2}$ processors away. (See Figure 3.) Phase 2 connects processors together that are $\frac{n}{4}$ processors away, but only in blocks of $\frac{n}{2}\times\frac{n}{2}$ processors. There are no connections between the blocks of $\frac{n}{2}\times\frac{n}{2}$ processors. Finally, phase k connects blocks of 2×2 processors with the grid pattern. (See Appendix B for the 16×16 connections.) In each phase, a processor has exactly two other processors connected to it. The port that is connected to the processor in the same row is named "horiz". The port that is connected to the processor in the same column is named "vert".

A phase is roughly equivalent to a recursive call. Phase 1 is run for a "call" for an n×n product, phase 2 is run for a "call" for an $\frac{n}{2}\times\frac{n}{2}$ product, and phase k is run for a "call" for a 2×2 product. The phases must be run in the proper order for a correct result. We will discuss the correct order later.

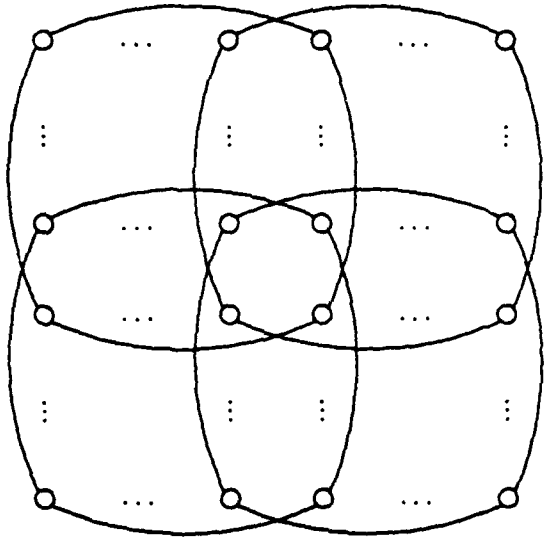


Figure 3: n×n connections

To keep track of "local variables" and the progress through the algorithm, a stack is implemented. A "stack frame" is composed of three elements. The top element of the frame contains a tag. This tag, an integer value of 1 or 2, represents which recursive call this frame is recording. The other two elements either contain two values waiting for a later multiply or contain the result of a previous multiply.

Communication from phase to phase is done using the feature of XX that identical declaration sections use the same memory allocation. All XX codes have the same prefix in their declarations. (See Figures 4 and 5 and Appendix A.) This yeilds an "unnamed common." The stack is part of this common area.

The other variables in the common area are "aele", "bele", "othera", "otherb", and "top". At the start of the a phase, "aele", and "bele" contain the corresponding values of A and B for the current product. The variables "othera" and "otherb" are used as local variables, and "top" points to the top of the stack. A larger value for "top" implies a larger stack. The value 0 implies an empty stack. This variable is assumed to be 0 at the start of phase 1.

```
code mmii; /* upper left and lower right */

ports vert, horiz;

begin
 int vert, horiz;
 int aele, bele, othera, otherb;
   /* aele and bele are the entries from A*B, the result is left in aele */
 int stack [ 24 ];
 sint top;

 /* send values */
 horiz <- aele;
 othera <- horiz;
 vert <- bele;
 otherb <- vert;

 /* multiply */
 /* start "recursion" */
 stack[top+1] := othera;
 stack[top+2] := otherb;
 stack[top+3] := 1;      /* this is first multiply */
 top := top + 3;

end.
```

Figure 4: XX code for processors in the upper left and lower right.

Each phase is divided into two distinct operations. The first is the communication part. The second is the computation and bookkeeping part. (Phase 1 to k-1 use the codes "mmii" and "mmij". Phase k uses the codes "mm2ii" and "mm2ij". See Figures 4 and 5 and Appendix A.) For the communication part, each processor sends its a_{ij} (aele) value out on the port "horiz". Similarly, the b_{ij} (bele) value is sent out on the port named "vert". The values that are received are placed in the variables "othera" and "otherb."

After the communication steps, $A_{i1}B_{1j}+A_{i2}B_{2j}$ must be evaluated. Remember that A_{11} is the upper left quarter of the 2×2 matrix of matrices or scalars. If it is a 2×2 matrix of matrices, the case for phases 1 to k-1, this is computed by doing two matrix products and a matrix add. Since these products can not be done in parallel, the data for one of these matrix products must be stacked. In both the upper left and the lower right, "aele" and "bele" contain the elements that need to be multipled. To reduce data movement, we then stack "othera" and "otherb". (See Figure 4.) In the lower left and the upper right, we must stack one of "aele" and "bele" and one of "othera" and "otherb". We arbitrarily choose to stack "othera" and "bele". (See Appendix A.) Since this is the first of the matrix products, the tag 1 is stacked.

After the the new stack frame has been added to the stack, it is time for the first "recursive call". Notice that the current phase is completed. We do not return to this phase because the setup for the "second" recursive call is done in phase k (2×2 connections). The work can be done there because no communication is required to set up for the "second" recursive call. Also, the matrix addition is performed in phase k for the same reason. The recursive calls are implemented by running the next phase. For example, the recursive calls associated with phase 1 are performed by running phase 2.

Consider the operation of phase k. Remember that the communication is for 2×2 matrix product of scalars. The communication takes place in the same manner as all other k-1 phases. After the communication, each processor contains all the data necessary to compute the c_{ij} for the 2×2 product. The processors in the upper left and lower right compute "aele*bele+othera*otherb". (See Figure 5.) The processors in the lower left and the upper right compute "aele*otherb+othera*bele". (See Appendix A.) The result is placed in "aele".

At this point, the recursion is terminated. We now simulate the returns. This is where matrix addition is done along with stack clean up. If the tag on the top stack frame is 2, the stack frame contains the result of a matrix multiply that needs to be added to the value in "aele". After the addition, the stack frame

```
code mm2ii; /* upper left and lower right */

ports horiz, vert;

begin
 int horiz, vert;
 int aele, bele, othera, otherb;
   /* aele and bele are the entries from A*B, the result is left in aele */
 int stack [ 24 ];
 sint top;
 bool second;

 /* send values */
 horiz  <- aele;
 othera <- horiz;
 vert <- bele;
 otherb <- vert;

 /* multiply */
 aele := aele * bele + othera * otherb;

 /* clean up stack and get ready for next communication phase */
 second := true;
 while second & (top > 0) do begin
  if stack[top] = 1 then begin
   /* only first multiply has been done */
   bele := stack[top-1];
   stack[top-1] := aele;  /* save result of first multiply */
   aele := stack[top-2];
   stack[top] := 2;  /* this is second multiply */
   second := false;  /* need to start a new phase */
  end else begin
   /* finished, just add */
   aele := aele + stack[top-1];
   top := top -3;
  end;
 end;

end.
```

Figure 5: XX code for 2×2 case.

is deleted from the stack. This is repeated until the stack is empty or the tag in the stack frame is a 1. When the stack is empty, the n×n product is stored in "aele".

When the tag in the top stack frame is a 1, we need to set up for the second recursive call for some recursive level. The level is unknown. The actions are always the same. The value in "aele" is the result of the first product. This needs to be put on the stack during the second recursive call. The values in the top stack frame must be put into "aele" and "bele". Also, the tag in stack frame must be changed from 1 to 2. With this done, it is time to change to the correct phase. This may be any of phases 2 to k.

The phases must be run in the correct order. Since there is no automatic control for running phases in Poker, the programmer must know the order. For the n×n product, the order is phase 1 followed by 2 sequences of phases for $\frac{n}{2}\times\frac{n}{2}$ product. For the $\frac{n}{2^j}\times\frac{n}{2^j}$ product, the order is phase $j+1$ followed by two sequences of the order for $\frac{n}{2^{j+1}}\times\frac{n}{2^{j+1}}$ product. For example, the 8×8 order is phases 1, 2, 3, 3, 2, 3, and 3. After the execution of all the phases, the result of the n×n matrix-matrix product is found in "aele".

4. The WAP Algorithm

To evaluate this recursive algorithm, we chose to implement and compare the algorithm for the Wavefront Array Processor(WAP)[4]. Although the WAP may not be a systolic array, the matrix product algorithm has data flow patterns typical in systolic algorithms.

The matrix product algorithm for the WAP uses n^2 processors connected in a n×n grid. The data is fed in along the top n processors and from the left n processors. The matrix A is arranged to enter column by column, starting with the first column. The matrix B is arranged to enter row by row, starting with the first row. (See Figure 6.) All processors execute identical procedures. The result, c_{ij}, is initialized to zero. A loop is executed n times that reads an A value from the left and a B value from above, multiplies them together, and adds the result to c_{ij}. The A and B values are sent to the right and down respectively. This

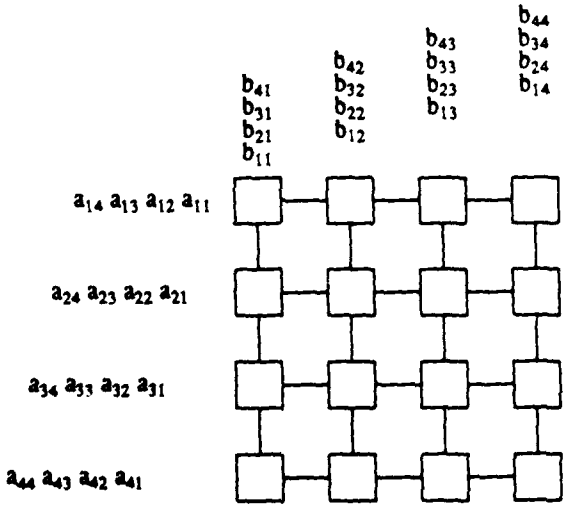


Figure 6: WAP organization

causes the upper left processor to be the first processor to start execution. As the data moves into the array, there is a wavefront of executing processors in a diagonal across the array. PE_{nn} cannot start processing until the data wavefront reaches it. With this delay and the compute time necessary to compute c_{nn}, PE_{nn} uses

$$t(n) = (4n-4)t_c+n(t_a+t_o)+nt_m$$

time, where the constants measure the same quantities as before, but for this algorithm. The recursive algorithm has a factor of two fewer communication steps. This can have an impact when t_c is the largest of the constants.

This algorithm was also implemented on the Poker system. It was modified to start with the data in the processors. The processors then circulate the data using end around connections to connect the last processor in a row or column to the first processor in the same row or column. (See Figure 7.) Notice that the matrix multiply is correctly computed if the a's and b's in Figure 6 were to enter the array in the reverse order. The reverse order is the result of a direct right shift and down shift of all the data using the end around connections.

The activity at each processor is divided up into two parts. The first is to circulate the data. (See Figure 8.) Each processor sends its a to the "right" and its b "down". Then, it repeats the a's coming in from

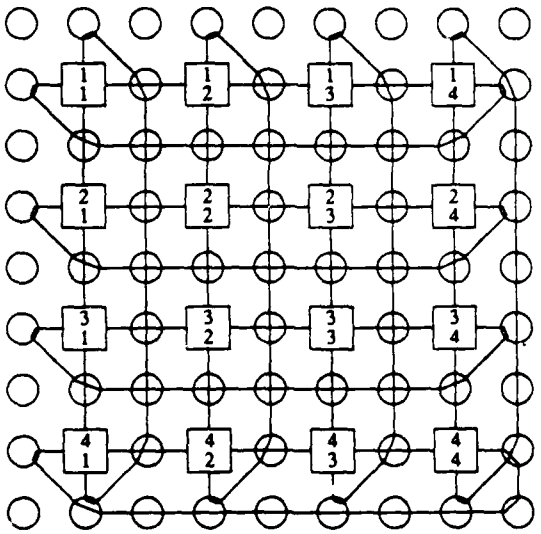


Figure 7: 4×4 connections for adapted WAP algorithm

the "left" and the b's from "up". After all the data has been passed on, all that remains is to compute its c_{ij}. This is just the accumulation of the a's times the b's in the order that they arrive at the processor. These a's and b's are sent on to the next processor in the same row and column respectively. The only exception to this is the last row and column. There is no need to send the A values "right" from the last column, or to send the B values "down" from the last row. This gives rise to three special codes, (See Appendix A.) one for the last row, the righmost column, and the lower right pe, $PE_{n,n}$.

5. The Modified WAP Algorithm

The direct implementation of the WAP algorithm followed the wavefront behavior of the original machine. With the flexability provided by then CHiP architecture, we can do some preprocessing on the

```
code matmul ( size );  /* WAP, all except last row and column */
ports  left, right, up, down;
begin
 int aele, bele, cele;
 int left, right, up, down;
 int size;
 sint indx, PEn, max;

 PEn := size;

 /* compute max of PEi, PEj */
 if PEi > PEj then
   max := PEi
 else
   max := PEj;
 /* start wave around */
 right <- aele;
 down  <- bele;
 for indx := 2 to max do begin
    if PEj >= indx then begin aele <- left; right <- aele end;
    if PEi >= indx then begin bele <- up; down <- bele end;
 end;
 /* do the multiply */
 for indx := 1 to PEn do begin
   aele <- left;
   right <- aele;
   bele <- up;
   down <- bele;
   cele := cele + aele * bele;
 end;
end.
```

Figure 8: XX code for WAP matrix product

rows of A and the columns of B so that all processors can start working immediately. Since they all start at the same time, they all stop at the same time, eliminating the $(2n-2)t_c$ time required for the wavefront to reach processor PE_{nn}.

The first phase of the algorithm does the preprocessing. Row i shifts the elements of A right by i-1 processors. Column j shifts the elements of B down by j-1 processors. The shifts assume wrap around, that is, processor PE_{in} is left of PE_{i1}. For this implementation, it was possible to put a direct connection between the source and destination processors. Figure 9 shows the 8×8 connections. (See Appendix A for the processor codes.)

After the preprocessing, each processor contains data required to start the sum of products immediatly. The sum variable is initialized with the product of the current a_{ik} and b_{kj}. The loop executed by each processor sends the current a_{ik} to the right, the current b_{kj} down and then adds to the sum variable the product of the a_{ik} and b_{kj} it just received from the left and above respectively. After n-1 iterations, the loop terminates.

The time used to compute the matrix product using the modifed WAP algorithm is

$$t(n) = t_r(n)+(2n-2)t_c+(n-1)(t_a+t_o)+nt_m$$

where $t_r(n)$ is the time for the preprocessing, and the constants measure the same quantities as before, but for this algorithm. Since $t_r(n)$ is a constant for our implementations, we can ignore it. For larger size problems, this cost may not be constant. For the 16×16 implementation, the preprocessing had to be done using two phases due to the number of connections required. (See Appendx C for the interconnection structure.)

n×n	recursive (ticks)	WAP (ticks)	Mod WAP (ticks)	n^3 (ticks)
2×2	2219	6458	4095	15054
4×4	7571	14065	7938	135814
8×8	18359	30696	15646	1150950
16×16	40385	70481	31354	9469030

Table 1: Running times on Poker 3.0

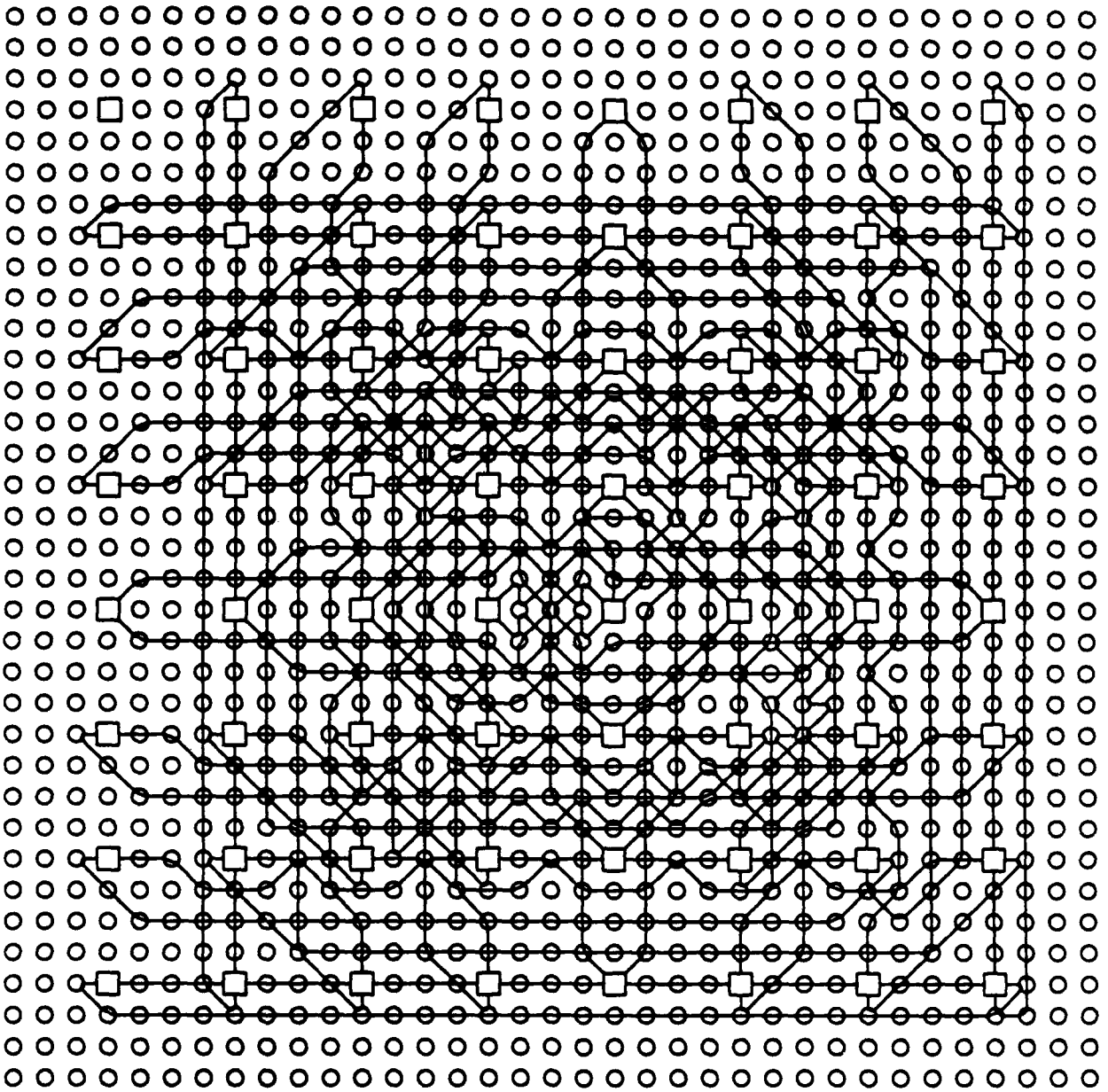


Figure 9: Modified WAP preprocessing connections for 8×8

6. Experimental Results

All of these algorithms were implemented for several values of n. All algorithms were run with two extra phases. The first extra phase loaded the test data into the processors from a file. The other extra phase dumped the results into a file. Since we were interested in the performance of these algorithms starting with the data in the processors, we ignored the time used in these extra phases. Table 1 gives a summary of the run times for these algorithms. One tick represents one microsecond on the 64 processor Pringle, the machine simulated by the Poker system. For any size with 64 or fewer processors, these times can be acheived by the Pringle. For the sizes with more than 64 processors, the times are arrived at by using the same ground rules used by the Pringle.

Because of the Pringle's rather antiquated technology, the speed-up comparisons are more relevant than the absoulte times. A 1 processor system was used to run the standard 3 loop, n^3 algorithm for all sizes of matrices. (See Appendix A for the code.) The speed-ups listed in Table 2 are using the sequential value divided by the parallel value. We can see the lack of overhead for the 2×2 recursive algorithm. Also, we can see the effect of the constant step in the modified WAP algorithm.

PE count	recursive	WAP	Mod WAP
4	6.78	2.33	3.68
16	17.94	9.66	17.11
64	62.70	37.50	73.56
256	234.47	134.35	302.00

Table 2: Speed ups

7. Summary

In designing algorithms for general purpose MIMD architectures, it is possible to adapt known algorithms. We have shown that this adaption may not produce the most efficient algorithm on the new architecture. We have shown two algorithms that had better performance than the adapted algorithm. One of these was a modifcation of the orignal algorithm taking advantage of the new architecture and original data placement. The other algorithm showed that designing a totally new algorithm may yield good results. Also, we have shown that the divide-and-conquer paradigm is useful for developing new algorithm.

Acknowledgements

I would like to thank Lawrence Snyder for his invaluable guidance and help, and W. Larry Ruzzo for suggesting the modified WAP algorithm.

References

1. Chern, M.Y. and Murata, T. "Efficient Matrix Multiplications on a Concurrent Data-Loading Array Processor", *Proc. 1983 Int. Conf. on Parallel Processing*, pp. 90-94.

2. Huang, K.H. and Abraham, J.A. "Efficient Parallel Algorithms for Processor Arrays", *Proc. 1982 Int. Conf. on Parallel Processing*, pp. 271-279.

3. Kung, H.T. "The Structure of Parallel Algorithms", *Advances in Computers, Vol. 19*, Academic, New York, 1980, pp. 65-111.

4. Kung, S.Y., Arun, K.S., Gal-Ezer, R.J., and Rao, D.V.B. "Wavefront Array Processor: Language, Architecture, and Applications", *IEEE Trans. Computers*, Nov. 1982, pp. 1054-1066.

5. Mead, C. and Conway, L. *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

5. Snyder, L. "Parallel Programming and the Poker Programming Environment", *Computer*, 17(7), pp. 27-36, July 1984.

6. Snyder, L. "Introduction to the Configurable, Highly Parallel Computer," *Computer*, 15(1), pp. 47-56, Jan 1982.

7. Strassen, V. "Gaussian elimination is not optimal," *Numerische Mathematik* 1969, 13, pp. 354-356.

Appendix A: XX codes

Recursive algorithm codes:

```
code mmij; /* lower right and upper left */

ports horiz, vert;

begin
 int horiz, vert;
 int aele, bele, othera, otherb;
    /* aele and bele are the entries from A*B, the result is left in aele */
 int stack [ 24 ];
 sint top;

 /* send values */
 horiz <- aele;
 othera <- horiz;
 vert  <- bele;
 otherb <- vert;

 /* multiply */
 /* start "recursion" */
 stack[top+1] := othera;
 stack[top+2] := bele;
 bele := otherb;
 stack[top+3] := 1;    /* this is first multiply */
 top := top + 3;

end.


code mm2ij; /* lower left and upper right */

ports horiz, vert;

begin
 int horiz, vert;
 int aele, bele, othera, otherb;
    /* aele and bele are the entries from A*B, the result is left in aele */
 int stack [ 24 ];
 sint top;
 bool second;

 /* send values */
 horiz <- aele;
 othera <- horiz;
 vert  <- bele;
 otherb <- vert;

 /* multiply */
 aele := aele * otherb + othera * bele;

 /* clean up stack and get ready for next communication phase */
 second := true;
 while second & (top > 0) do begin
```

```
  if stack[top] = 1 then begin
   /* only first multiply has been done */
   bele := stack[top-1];
   stack[top-1] := aele;  /* save result of first multiply */
   aele := stack[top-2];
   stack[top] := 2;  /* this is second multiply */
   second := false;  /* need to start a new phase */
  end else begin
   /* finished, just add */
   aele := aele + stack[top-1];
   top := top -3;
  end;
 end;

end.
```

WAP codes:

```
code mmlow ( size ); /* WAP, last row */

ports  left, right, up, down;

begin
 int aele, bele, cele;
 int left, right, up, down;
 int size;
 sint indx, PEn;

 PEn := size;

 /* start wave around */
 right <- aele;
 down  <- bele;
 for indx := 2 to PEn do begin
   if PEj >= indx then begin aele <- left; right <- aele end;
   if PEi >= indx then begin bele <- up; down <- bele end;
 end;
 /* do the multiply */
 for indx := 1 to PEn do begin
  aele <- left;
  right <- aele;
  bele <- up;
  cele := cele + aele * bele;
 end;
end.


code mmrig ( size );  /* WAP, rightmost column */

ports  left, right, up, down;

begin
 int aele, bele, cele;
 int left, right, up, down;
 int size;
```

```
 sint indx, PEn;

 PEn := size;

 /* start wave aroun
 right <- aele;
 down  <- bele;
 for indx := 2 to PEn do begin
   if PEj >= indx then begin aele <- left; right <- aele end;
   if PEi >= indx then begin bele <- up; down <- bele end;
 end;
 /* do the multiply */
 for indx := 1 to PEn do begin
  aele <- left;
  bele <- up;
  down <- bele;
  cele := cele + aele * bele;
 end;
end.


code mmlr ( size );  /* WAP, lower right pe, PEn,n */

ports  left, right, up, down;

begin
 int aele, bele, cele;
 int left, right, up, down;
 int size;
 sint indx, PEn;

 PEn := size;

 /* start wave around */
 right <- aele;
 down  <- bele;
 for indx := 2 to PEn do begin
   if PEj >= indx then begin aele <- left; right <- aele end;
   if PEi >= indx then begin bele <- up; down <- bele end;
 end;
 /* do the multiply */
 for indx := 1 to PEn do begin
  aele <- left;
  bele <- up;
  cele := cele + aele * bele;
 end;
end.
```

Modified WAP codes:

```
code aroute; /* first column except processor 1,1 */

ports ain, aout;

begin
```

```
  int aele, bele;

  /* communicate */
  aout <- aele;
  aele <- ain;

end.


code broute; /* first row except processor 1,1 */

ports bin, bout;

begin
  int aele, bele;

  /* communicate */
  bout <- bele;
  bele <- bin;

end.


code route; /* all processors except first row and first column */

ports ain, aout, bin, bout;

begin
  int aele, bele;

  /* communicate */
  aout <- aele;
  aele <- ain;
  bout <- bele;
  bele <- bin;

end.


code matmul ( PEn );

ports  left, right, up, down;

begin
  int aele, bele, cele;
  int PEn;
  sint indx;

  /* do the multiply */
  cele := aele*bele;
  for indx := 2 to PEn do begin
    right <- aele;
    aele <- left;
    down <- bele;
    bele <- up;
    cele := cele + aele * bele;
  end;
```

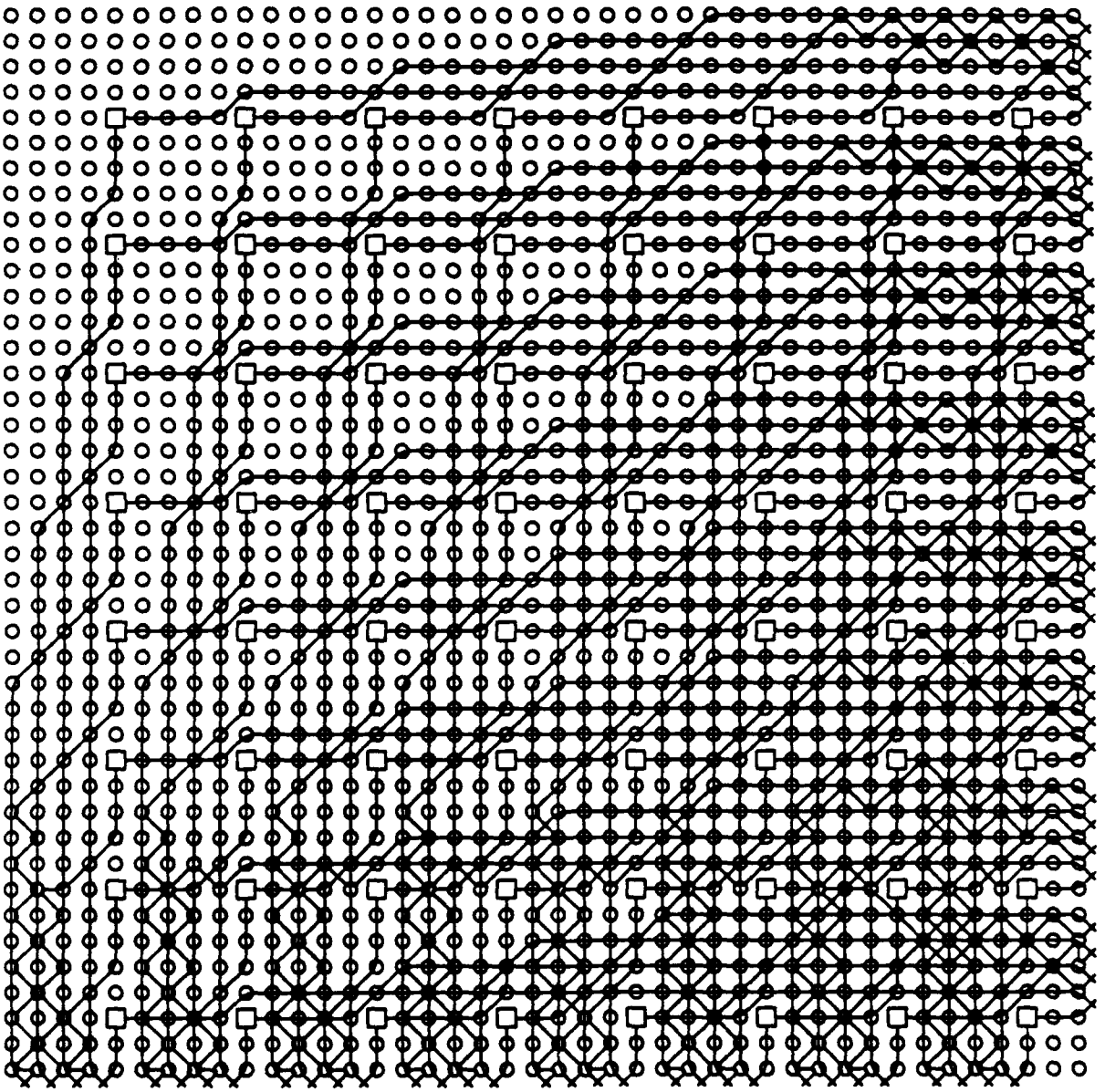
end.

Sequential code:

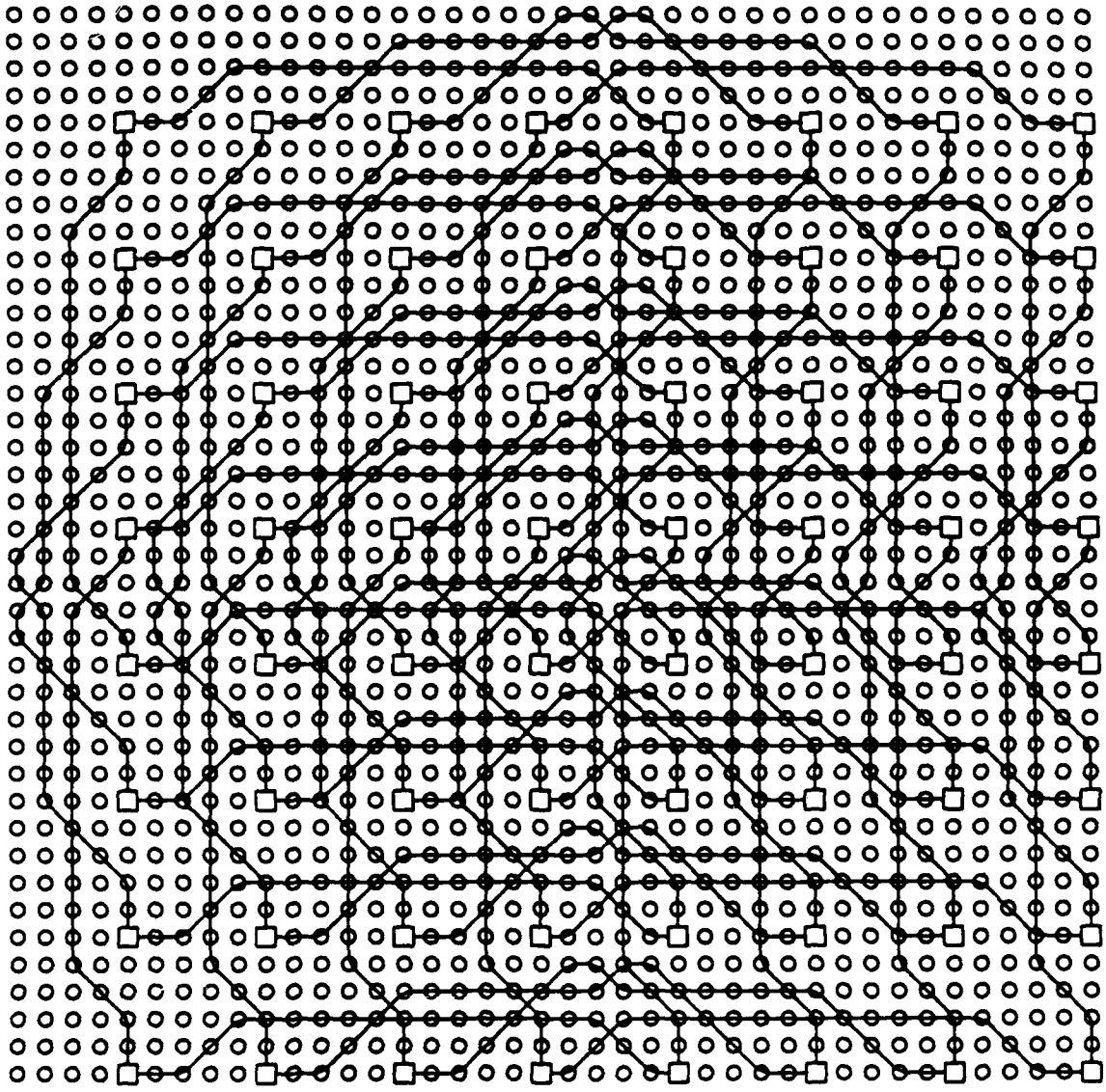
```
code matmul; /* sequential version for comparison */

begin
 int aele[16,16], bele[16,16], cele[16,16];
 sint size;
 sint i, j, k;

 /* multiply */
 for i := 1 to size do
  for j := 1 to size do begin
   cele[i,j] := aele[i,1] * bele[1,j];
   for k := 2 to size do
    cele[i,j] := cele[i,j] + aele[i,k] * bele[k,j];
  end;

end.
```

Appendix B: 16×16 connections

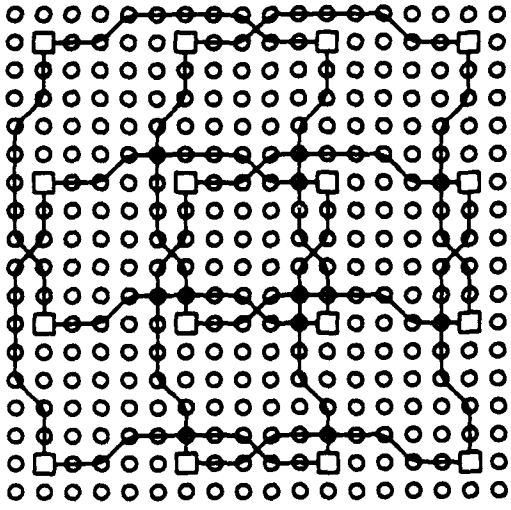
The connection structure for the 16×16 is not immediately obvious. For completeness, we are including all the interconnection structures for the 16×16 product. We will show enough of each structure so that the full 16×16 can be reconstructed by use of replication and reflection.



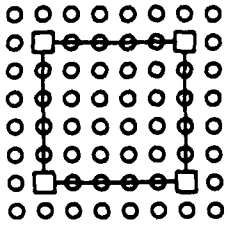
16×16 upper left corner.



8×8



4x4



2x2

Appendix C: Modified WAP 16×16 route connections

The following is enough of the connection structure to reconstruct the complete structure using reflections and rotations. This phase routed only the rows of A. A second phase, a rotation of the first phase, was used to route the columns of B.