Department of Information and Computer Science

University of California at Irvine

Irvine, CA 92717

Executing Matrix Multiply on a Process Oriented Dataflow Machine

Lubomir Bic
Mark D. Nagel
John M.A. Roy

April 1990

Technical Report #90-08

Abstract

The Process-Oriented Dataflow System (PODS) is an execution model that combines the von Neumann and dataflow models of computation to gain the benefits of each. Central to PODS is the concept of array distribution and its effects on partitioning and mapping of processes.

In PODS arrays are partitioned by simply assigning consecutive elements to each processing element (PE) equally. Since PODS uses single assignment, there will be only one producer of each element. This producing PE *owns* that element and will perform the necessary computations to assign it. Using this approach the filling loop is distributed across the PEs. This simple partitioning and mapping scheme provides excellent results for executing scientific code on MIMD machines. In this way PODS allows MIMD machines to exploit vector and data parallelism easily while still providing the flexibility of MIMD over SIMD for multi-user systems.

In this paper, the classic matrix multiply algorithm, with 1024 data points, is executed on a PODS simulator and the results are presented and discussed. Matrix multiply is a good example because it has several interesting properties: there are multiple code-blocks; a new array must be *dynamically* allocated and distributed; there is a loop-carried dependency in the innermost loop; the two input arrays have different access patterns; and the sizes of the input arrays are not known at compile time. Matrix multiply also forms the basis for many important scientific algorithms such as: LU decomposition, convolution, and the Fast-Fourier Transform.
The results show that PODS is comparable to both Iannucci's Hybrid Architecture and MIT's TTDA in terms of overhead and instruction power. They also show that PODS easily distributes the work load evenly across the PEs. The key result is that PODS can scale matrix multiply in a near linear fashion until there is little or no work to be performed for each PE. Then overhead and message passing become a major component of the execution time. With larger problems (e.g., ≥16k data points) this limit would be reached at around 256 PEs.

Keywords: single assignment, dataflow, multiprocessor, scientific programming, matrix multiply

1. Introduction 1

2. The Process-Oriented Dataflow System 2
   2.1. Dataflow and Hybrid Architectures 2
   2.2. PODS Overview 4

3. Dataflow Graph Partitioning 7
   3.1. Data and Functional Parallelism 9
      3.1.1. Scalars 10
      3.1.2. Arrays 10
      3.1.3. Code Blocks 11
   3.2. Partitioning Programs into Code-Blocks 12
   3.3. Partitioning Arrays and Loops 13
   3.4. Mapping Subcompact Processes 15

4. Simulation Results 18
   4.1. Simulation Parameters 18
   4.2. Speedup 18
   4.3. Load Balance 20
   4.4. Comparisons 21
      4.4.1. Hybrid Architecture 21
      4.4.2. Alfalfa 22

5. Conclusions 22

References 23

Appendix A: Matrix Multiply in PODS 26

Appendix B: Range Generators 29
   B.1. Objective and Usage 29
   B.2. Boundary Table 30
   B.3. Algorithm 31

graphs into communicating processes, however, were too simplistic, concentrating on only functional parallelism. In scientific code, most parallelism comes from loops iterating over large data structures (i.e., data parallelism). We have addressed this issue in subsequent studies [Bic90, BNR89a, BNR89b] by showing that, for languages based on the single-assignment principles (declarative languages), a simple automatic partitioning of arrays exposes significant parallelism that can be exploited at run-time.

Based on those studies and these reported herein, we describe an approach to automatically transform Id Nouveau programs into sets of processes communicating with one another through messages and accessing a global address space containing distributed data structures. The performance of this approach when executing the matrix multiply algorithm is examined to show how the most common scientific code would operate. Matrix multiply forms the basis for many important scientific algorithms such as LU decomposition, convolution, and Fast-Fourier transforms. The paper is organized as follows. Section 2 examines the PODS execution model and architecture. Section 3 explains dataflow graph partitioning and mapping in PODS, and introduces the matrix multiply example. Section 4 discusses the simulator and the simulation results for matrix multiply. Finally, Sections 5 and 6 present the conclusions and references, respectively.

2. The Process-Oriented Dataflow System

2.1. Dataflow and Hybrid Architectures

Since Dennis first described the first dataflow execution model [Den75], many architecture designers have attempted to apply the model to real systems. Dataflow is attractive because all parallelism in a program is exposed for potential concurrent execution. In spite of the elegance of the model, dataflow is not widely used after more than twenty years of research. The focus has instead turned to the evolution of modern systems by extending them with dataflow techniques. The results of research in this area include

hybrid systems using large-grain or macro dataflow [Bab84, B&E87, DFL89, Ian88, Kap86, L&G86, S&H87].

Iannucci [Ian88] has reported on a hybrid dataflow / von Neumann architecture. This approach is similar to PODS in its use of Id Nouveau as the input language and split-phased structure access. However the Iannucci approach uses a finer grain scheduling approach, called scheduling quanta (SQ). An SQ of two to three instructions is desirable for Iannucci's approach, and each iteration of a loop is a new SQ. In PODS, however, the natural decomposition of the program is used and SPs are allowed to run-in-place, thus reducing overhead. Another difference is in data structure distribution. There is no mechanism for spreading iterations of a single loop across process in Iannucci's approach. Combining data structure distribution with loop distribution is a central goal in PODS. Finally Iannucci's model required a special purpose architecture capable of fast context switching among very small SQs. PODS tries to generate SPs large enough to produce good computation-communication ratios on available distributed memory multiprocessors. Certainly PODS would benefit from a tailored architecture, but the model itself is not restricted to such.

In [GH89], Goldberg and Hudak presented Alfalfa, a system similar at a high level to PODS. They have implemented the ALFL functional programming language and run-time system on an Intel iPSC hypercube using what they call *serial combinators*. Serial combinators are similar to PODS SPs in that they are sequential threads that execute on a von Neumann processor. The run-time system handles thread creation and distribution. The main focus of their work is the study of the effects of dynamic scheduling (diffusion scheduling) of parallel threads of execution. They show that diffusion scheduling works well in many cases, however, they have not addressed the problem of distributing large data structures such as arrays. This is illustrated through the relatively poor performance achieved with the matrix multiply algorithm.

3.1. Data and Functional Parallelism

In PODS one objective is to exploit data parallelism efficiently without losing functional parallelism. On distributed memory machines this can be difficult to achieve. By using the above heuristics and the single-assignment principle, data parallelism and functional parallelism can co-exist in distributed memory. In most scientific code the data parallelism is greater than the functional parallelism. The Translator/Partitioner understands this and optimizes its transformations to exploit data parallelism. Currently, functional parallelism is exploited by randomly distributing functional code-blocks, but future

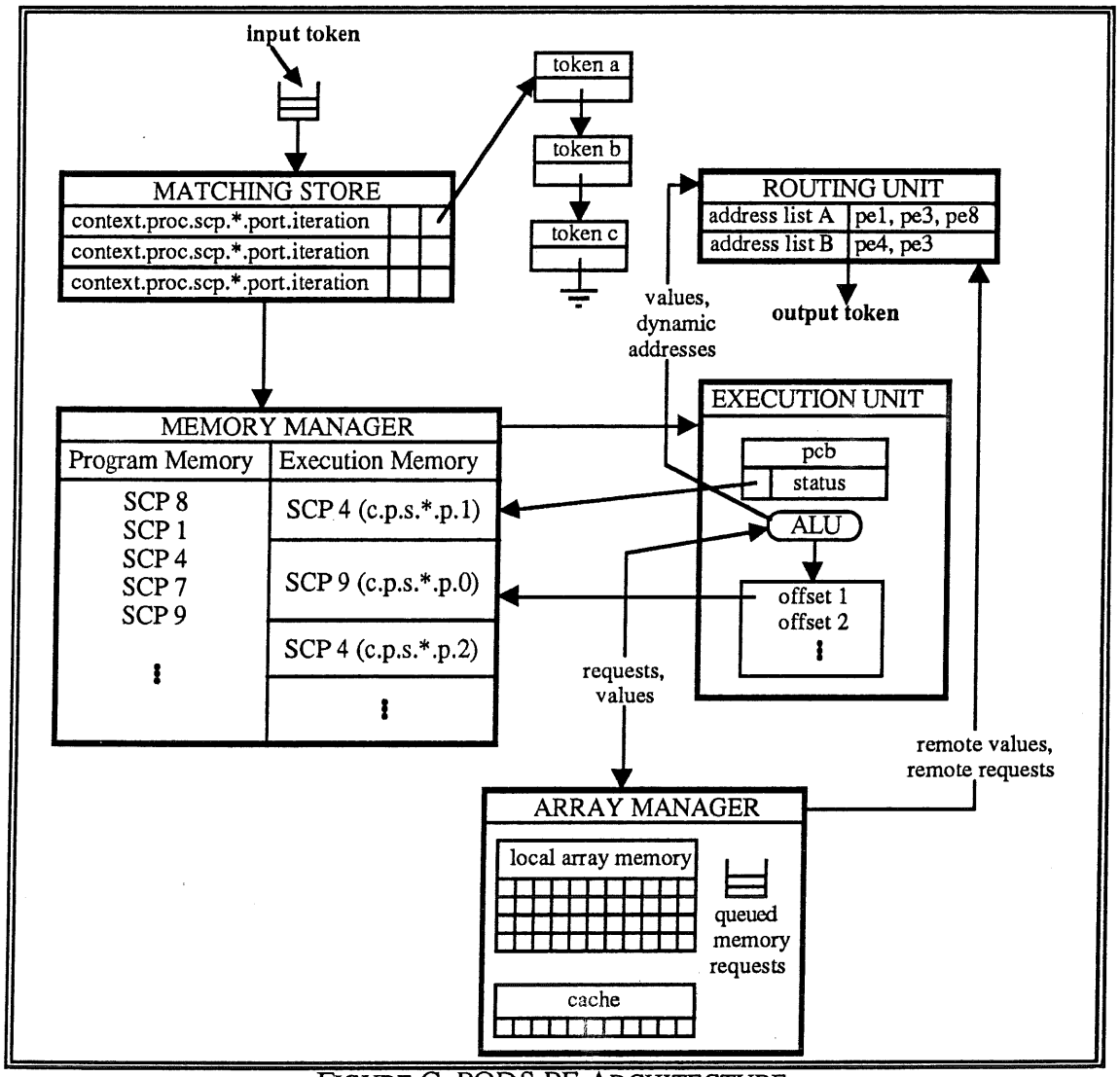


FIGURE C. PODS PE ARCHITECTURE

versions of the Translator/Partitioner may use more intelligent techniques.

Code-blocks are one kind of high-level object manipulated by PODS. Two others are scalars and arrays. Scalars are essentially abstractions representing communication channels (implemented as tagged tokens). Operations over arrays provide data parallelism and the execution of functional code blocks provides functional parallelism.

3.1.1. Scalars

For scalar variables no explicit partitioning or mapping is necessary—a scalar is produced locally and then automatically sent in the form of a tagged token to the consuming PE by matching the tag with the corresponding remote operator. In the case that the token is sent to the same SP that produced it, no actual routing is done—the token is simply stored directly in the destination instruction's operand list. Thus, during the sections of code where an SP does no external communication, the SP instructions are nearly identical to typical von Neumann instructions. This allows PODS to use well-known optimization techniques on large portions of the code within SPs.

3.1.2. Arrays

For arrays, we exploit the single-assignment principle of I-structures [ANP89]. This guarantees that only one instruction will write to an array element—it is the *producer* of that data. We use this fact by attempting to map each array element onto the same PE as its producer instruction. This distributes data across all of the PEs participating in the computation.

When an array is allocated, the allocating PE broadcasts a message telling each PE to reserve an appropriate amount of local memory. Since each PE knows the size of the newly allocated array (from the message), it can determine what its area-of-responsibility for that array is. As a result, each PE derives a *boundary table* for the array. The boundary table will be used later to control a *range generator* that generates only those loop indices needed by each PE. The boundary table is a simple list of ranges of elements that the PE is

3.3. Partitioning Arrays and Loops

As discussed earlier, each array is divided into pages of fixed size that are distributed across the PEs. The loop body accessing array elements is identical for each PE, so a copy of the loop body code is distributed to each PE. The loop body code uses local data (the boundary table) to produce the data for its array partition. More precisely, the partitioning of a loop takes place as follows:

- Partition data by segmenting each array into pages of some fixed size. These pages are then sequentially grouped together into what are called *superpages*, one superpage per PE.
- Partition loop control by assigning to each PE the responsibility for updating the elements in all the array pages it contains in its local memory.

In ideal circumstances all data is written to locally, but in some cases program structure leads to inevitable remote data writes. Still, each PE is assigned a set of data that no other PE is expected to write. Note that it is not always possible to determine which element is being updated by an assignment statement at compile time. Consider the loop below:

```
      DO 10 K = 1,N
         A(F(K)) = B(G(K))
   10 CONTINUE
```

The functions F and G make it impossible to determine which element a given K will correspond to at compile time. In this case each PE must calculate F(K) for *all* values of K to determine whether an element of A is inside its area of responsibility. It should also be noted that arrays are single assignment and that the F(K) must be well behaved (one-to-one) over the range of K, otherwise a single assignment run-time error will occur.

As a simple example of this partitioning method, suppose we have a multiprocessor with 4 PEs. Using a page size of 32 elements, and 3 arrays A, B, and C, each of size 100.

PE 0, 1, and 2 will each contain a single page of each array. PE 3 will contain a partial page (4 elements) of each array. Consider the following loop:

```
      DO 10 I = 1,100
10      A(I) = B(101-I) + C(I)
```

All four processors begin executing simultaneously—PE 0 fills A(1..32), PE 1 fills A(33..64), PE 2 fills A(65..96), and PE 3 fills A(97..100). Note that for most of the loop, each processor must access elements of array B that lie on a different processor than the executing processor. Each one of these remote accesses requires a transfer of data from the producing PE to the consuming PE, an operation that is relatively expensive on all current distributed memory multiprocessors. It will never be possible to remove the need for remote accesses from distributed computations, so we are instead using a technique to diminish their effect on the overall computation time—remote access caching.

Remote access caching takes advantage of the fact that in PODS single-assignment enforcement ensures that no array element will be written multiple times. As a result, PEs may cache data that have been recently accessed without actively maintaining cache coherency. In the partitioning scheme defined earlier, each PE stores a portion of each array, with each portion consisting of fixed size pages. With remote access caching, reference to a remote array element causes retrieval of the entire page containing that element. The remote page is then stored locally and checked first in future references to that page. Due to locality of reference in many algorithms, it is likely that the same PE will need elements from that page in the near future, so potentially many remote accesses may be avoided. If the next requested element was not available at the time the page was originally cached, then another remote access transfering the same page (updated with the requested element) will be required. The performance enhancement achieved through remote access caching is detailed in [BNR89a].

program there may be other activities going on at the same time that would then be able to use the execution unit during these idle periods.

Number of PEs	Average EU Utilization	Standard Deviation	Mean EU Busy Period (μsec)	Standard Deviation
1	1.00	0.001	2,349.67	0.00
2	1.00	0.002	322.36	4.17
4	0.96	0.002	235.65	1.90
8	0.82	0.001	205.35	0.76
16	0.69	0.001	191.71	1.61
32	0.51	0.001	181.13	0.06
64	0.46	0.001	175.39	0.41

TABLE B. AVERAGE EXECUTION UNIT UTILIZATION AND BUSY PERIOD.

4.4. Comparisons

4.4.1. Hybrid Architecture

Iannucci used a dynamic instruction count comparison for his hybrid approach against the Tagged-Token Dataflow Architecture (TTDA) [A&N87]. One of his benchmarks was a 10x10 matrix multiply. Iannucci noted that TTDA instruction were strictly more powerful than hybrid instruction because of forking, yet the instruction counts were comparable to first order. Below is a table that includes PODS instruction counts for a 64 PE system (most overhead) running a 10x10 Matrix Multiply, as well as the hybrid and TTDA counts.

Iannucci stated that the reason comparable numbers of hybrid and TTDA instructions were executed was because the hybrid architecture exploited the sequential nature of programs to reduce overhead. We agree completely, and the fact that PODS instructions are on the same level as hybrid instructions and require even fewer instructions (even on a 64 PE system) indicates that PODS requires less overhead than either system when executing matrix multiply-like algorithms.

PE 0, 1, and 2 will each contain a single page of each array. PE 3 will contain a partial page (4 elements) of each array. Consider the following loop:

```
      DO 10 I = 1,100
10      A(I) = B(101-I) + C(I)
```

All four processors begin executing simultaneously—PE 0 fills A(1..32), PE 1 fills A(33..64), PE 2 fills A(65..96), and PE 3 fills A(97..100). Note that for most of the loop, each processor must access elements of array B that lie on a different processor than the executing processor. Each one of these remote accesses requires a transfer of data from the producing PE to the consuming PE, an operation that is relatively expensive on all current distributed memory multiprocessors. It will never be possible to remove the need for remote accesses from distributed computations, so we are instead using a technique to diminish their effect on the overall computation time—remote access caching.

Remote access caching takes advantage of the fact that in PODS single-assignment enforcement ensures that no array element will be written multiple times. As a result, PEs may cache data that have been recently accessed without actively maintaining cache coherency. In the partitioning scheme defined earlier, each PE stores a portion of each array, with each portion consisting of fixed size pages. With remote access caching, reference to a remote array element causes retrieval of the entire page containing that element. The remote page is then stored locally and checked first in future references to that page. Due to locality of reference in many algorithms, it is likely that the same PE will need elements from that page in the near future, so potentially many remote accesses may be avoided. If the next requested element was not available at the time the page was originally cached, then another remote access transfering the same page (updated with the requested element) will be required. The performance enhancement achieved through remote access caching is detailed in [BNR89a].

3.4. Mapping Subcompact Processes

SPs may be mapped in either of two ways: (1) via a simple hash function for functional code-blocks, or (2) via array distribution and range generators for loops. Array distribution controls a majority of the mapping and provides the bulk of the parallelism in scientific code. The hash function used to distribute functional code-blocks is:

$$\sum_i (SP_i + I_i) \textbf{ mod } N$$

where SP_i = a subcompact process ID, I_i = an iteration number, and N = total number of PEs. The subscript i refers to the stack of contexts leading to the current context.

This function provides a fairly random distribution, which in turn tends to generate a balanced work load (as we demonstrate later). Given more information, a more complex and possibly better distribution function may be used, but the simple approach achieves acceptable results without wasting interconnect bandwidth in order to maintain global state information.

Array distribution driven mapping is more complex. As noted above, the PODS Translator/Partitioner inserts instructions into the code-blocks to distribute a loop. In the case of matrix multiply, the loop that should be distributed is the *j*-loop. Different parts of the same loop will execute on different PEs (e.g., $i = 1, j = 10$ may execute on a different PE than $i = 1, j = 345$). The mapping depends on how the C array is distributed since C is the *master* array of the loop, i.e., it is the array being filled. The function of the range generator is to produce only those indices for which its PE is responsible, see Appendix B.

Figure D illustrates the distribution of SPs across four PEs. The curved lines represent broadcasts, the straight lines represent execution time, and the bold lines correspond to the comments on the right-hand side of the figure. For this example assume the matrix multiply starts out on PE 2. There SP 0 begins execution, and encounters the "ALLOCATE C" instruction. This instruction initiates a broadcast message to the other PEs. Upon receipt of this message, each PE allocates its portion of the array. Next, SP 0

generates and broadcasts the first value for *i*. Note that SP 0 does not have a range generator, thus it will generate *all i*-indices.

Each remote PE that receives an activating token (value 0) instantiates SP 1. SP 1 *does* have a range generator, so it will process only those indices for which the current PE is responsible. Thus a number of PEs quickly execute essentially empty SPs because they have no elements for which they are responsible when *i* is 0. In this case, PE 3 is the only PE with operations to perform when *i* is 0. PE 3 executes SP 1, which spawns the *k*-loop locally (the fact that the loop is local was determined at compile time). The *k*-loop is a simple loop that generates a vector dot product and returns the result to its parent SP. The *j*-loop may now continue with the *j* values for which it is responsible when *i* is 0.

In parallel with the execution of the first iteration of the *i*-loop, the original SP 0 continues generating and broadcasting successive values for *i* . This will cause new ready SPs to queue up in remote PEs. As other SPs block waiting for tokens, these new SPs will be selected for execution by the scheduler.

Once the *k*-loop starts, it will access remote pages from different PEs as necessary. This is where the existence of the remote access cache becomes important—a large number of reads will access the local array cache rather than causing a remote read.

program there may be other activities going on at the same time that would then be able to use the execution unit during these idle periods.

Number of PEs	Average EU Utilization	Standard Deviation	Mean EU Busy Period (μsec)	Standard Deviation
1	1.00	0.001	2,349.67	0.00
2	1.00	0.002	322.36	4.17
4	0.96	0.002	235.65	1.90
8	0.82	0.001	205.35	0.76
16	0.69	0.001	191.71	1.61
32	0.51	0.001	181.13	0.06
64	0.46	0.001	175.39	0.41

TABLE B. AVERAGE EXECUTION UNIT UTILIZATION AND BUSY PERIOD.

4.4. Comparisons

4.4.1. Hybrid Architecture

Iannucci used a dynamic instruction count comparison for his hybrid approach against the Tagged-Token Dataflow Architecture (TTDA) [A&N87]. One of his benchmarks was a 10x10 matrix multiply. Iannucci noted that TTDA instruction were strictly more powerful than hybrid instruction because of forking, yet the instruction counts were comparable to first order. Below is a table that includes PODS instruction counts for a 64 PE system (most overhead) running a 10x10 Matrix Multiply, as well as the hybrid and TTDA counts.

Iannucci stated that the reason comparable numbers of hybrid and TTDA instructions were executed was because the hybrid architecture exploited the sequential nature of programs to reduce overhead. We agree completely, and the fact that PODS instructions are on the same level as hybrid instructions and require even fewer instructions (even on a 64 PE system) indicates that PODS requires less overhead than either system when executing matrix multiply-like algorithms.

Hybrid Instructions	TTDA Instructions	PODS Instructions
23,569	20,118	16,467

TABLE C. DYNAMIC INSTRUCTION COUNTS FOR 10X10 MATRIX MULTIPLY.

4.4.2. Alfalfa

The Alfalfa system [GH89] is mainly concerned with different dynamic scheduling techniques and does not address the problem of distributing large data structures, such as arrays. They achieve some impressive results for problems involving little to no data communication, however, for matrix multiply, they see poor speedup results. They claim that this is due to the slow message passing time of the iPSC, but our system shows that a data cache combined with simple scheduling can help overcome the long latencies associated with accessing remote data.

5. Conclusions

PODS is a new execution model based on the concept of macro-dataflow. As such, the benefits of dataflow can be realized on a typical loosely-coupled MIMD computer. By simulating such a machine, we have shown that the model shows much promise for the efficient execution of declarative programs. In addition, the use of declarative programming languages will greatly reduce the development time for programs since the programmer will be released from explicitly managing parallelism.

The mechanism for distributing arrays in PODS not only allows for larger arrays than normally available in such machines, but it also takes advantage of locality of reference by keeping recently used parts of remote data in a cache. Matrix multiply is used as a demonstrative example and is used as a performance measure. Matrix multiply is a good measure because it has several interesting properties:

- there are multiple code-blocks
- a new array must be *dynamically* allocated and distributed

- there is a loop-carried dependency in the innermost loop
- the two input arrays, A and B, have different access patterns
- the sizes of the input arrays are not known at compile time

Matrix multiply also forms the basis for many important scientific algorithms such as: LU decomposition, convolution, and the Fast-Fourier Transform.

The results show that PODS is comparable to both the Hybrid Architecture and the TTDA in terms of overhead and instruction power. In addition, PODS is more capable of exploiting parallelism when large data structures must be shared among PEs than other similar systems, such as Alfalfa. The results also show that PODS distributes the work load evenly across the PEs. The key result is that PODS can scale matrix multiply in a near linear fashion until there is little or no work to be performed for each PE. Then overhead and message passing become a major component of the execution time. With larger problems (e.g., ≥16k data points) this limit would be reached around 256 PEs.

PODS thus allows MIMD machines to exploit vector and data parallelism simply while still providing the flexibility of MIMD over SIMD for multi-user systems.

References

[ACM88] Arvind, D. E. Culler, G. K. Maa. Assessing the Benefits of Fine-Grained Parallelism in Dataflow Programs. *MIT Technical Report Computation Structures Group Memo 279* (June 1988), Laboratory for Computer Science, MIT.

[A&E88] Arvind, K. Ekanadham. Future Scientific Programming on Parallel Machines. *J. Parallel Dist. Comp. V5, n5* (1988), pp. 460-493.

[A&N87] Arvind, R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *MIT Technical Report Computation Structures Group Memo 271* (March 1987), Laboratory for Computer Science, MIT.

[ANP89] Arvind, R. S. Nikhil, K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM TOPLAS V11, n4* (1989), pp. 598-632.

[Bab84] R. G. Babb. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer* (July 1984), pp. 55-61.

[Bic87] L. Bic. A Process-Oriented Model for Efficient Execution of Dataflow Programs. *Proc. 7th Int'l Conf. on Distributed Computing Systems* (1987).

[Bic90] L. Bic. A Process-Oriented Model for Efficient Execution of Dataflow Programs. *Journal of Dist. and Parallel Computing VMarch*, (1990).

[BNR89a] L. Bic, M. D. Nagel, J. M. A. Roy. Automatic Data/Program Partitioning Using the Single Assignment Principle. *Supercomputing '89* (1989), pp. 551-556.

[BNR89b] L. Bic, M. D. Nagel, J. M. A. Roy. On Array Partitioning in PODS. To appear in *Report on the 1989 Dataflow Workshop*. L. Bic, J. Gaudiot, Eds. (Prentice-Hall, 1989).

[B&E87] R. Buehrer, K. Ekanadham. Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution. *IEEE Trans. Comp. VC-36, n12* (1987), pp. 1515-1521.

[DFL89] D. DeForest, A. Faustini, R. Lee. Hyperflow. *The Third Conference on Hypercube Concurrent Computers and Applications* (1989), pp. 482-488.

[Den75] J. B. Dennis. First Version of a Dataflow Procedure Language. *Machine Tech. Memorandum 61,* MIT, Cambridge, MA.

[Dun88] T. H. Dunigan. Performance of a Second Generation Hypercube. *Technical Report ORNL/TM-10881* (November 1988), Oak Ridge National Laboratory.

[GH89] B. Goldberg, P. Hudak. Implementing Functional Programs on a Hypercube Multiprocessor. *The Third Conference on Hypercube Concurrent Computers and Applications* (1989), pp. 489-504.

[Ian88] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Ph.D. Thesis (1988), MIT.

[Kap86] I. Kaplan. A Large-Grain Dataflow Architecture. *Workshop on Future Directions in Computer Architecture and Software* (1986), pp. 131-138.

[L&G86] J. W. Liu, A. Grimshaw. A Distributed System Architecture Based on Macro Dataflow Model. *Workshop on Future Directions in Computer Architecture and Software* (1986), pp. 155-162.

[Mac87] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools*, MIT Press, Cambridge, MA, 1987.

[ANP87b] R. S. Nikhil. Id World Reference Manual (for Lisp Machines). *MIT Technical Report* (April 1987), Laboratory for Computer Science, MIT.

[Nik88] R. S. Nikhil. ID Reference Manual - Version 88.1. *MIT Technical Report Computation Structures Group Memo 284* (August 1988), Laboratory for Computer Science, MIT.

[Pap88] G. M. Papadopoulos. Implementation of a General-Purpose Dataflow Multiprocessor. *Technical Report TR-432* (August 1988), MIT Laboratory for Computer Science.

[R&P88] A. Rogers, K. Pingali. Process Decomposition Through Locality of Reference (August 1988), Department of Computer Science, Cornell University.

[R&P89] A. Rogers, K. Pingali. Compiling Programs for Distributed Memory Architectures. *4th Hypercube Concurrent Computers & Applications Conference* (1989).

[S&H87] B. Shirazi, A. R. Hurson. A Large/Fine Grain Parallel Dataflow Model and its Performance Evaluation. *1987 National Computer Conference* (1987), pp. 119-126.

Appendix A: Matrix Multiply in PODS

The Id Noveau code below was used in these simulations. It was first compiled using the Sun Common Lisp based Id World, version number 4.1. This produced three different code-blocks for hypothetical GITA machine. These code-blocks were inserted into the PODS Translator/Partitioner, which translates the GITA assembly code into PODS assembly code and partitions the code-blocks using the heuristics described earlier to generate the SPs below.

```
%%% Matrix Multiply
Def MM A B = {(l1,u1), (l2,u2) = 2D_bounds A;
              C = i_matrix ((l1,u1), (l2,u2));
              In
              { For i <- l1 To u1 Do
                { For j <- l2 to u2 Do
                  s = 0;
                  C[i,j] =
                    { For k <- l1 To u1 Do
                        Next s = s + A[i,k] * B[k,j];
                      Finally s
                    }
                };
                Finally C
              }
            };
```

The files below are the exact inputs that were used to run matrix multiply on the simulator. The outer-most loop (*i*-loop) code is in MM-0, the *j*-loop is MM-1, and the *k*-loop is MM-2. The assembly code is read as follows: (1) the instruction number, within the SP; (2) the instruction to be executed; (3) the number of arguments this instruction takes; (4) a list of the constant arguments to the instruction; (5) a list of local destinations (i.e., within this SP); (6) the routes by which this instruction communicates with other SPs; (7) an optional comment based on the Id World translation. Destination lists are surrounded by "[]", route lists by "()", and comments by "{}".

To clarify this, consider instruction 9 in SP MM-0 below. This LE (less than or equal to) operation take two arguments, one of which is sticky (port 1). A sticky port is one that retains a value forever once the value arrives. Instruction 9 will send its result to

instruction 10, port 0. Instruction 9 has no routes and no comment. The reason an instruction may have two routes is due to the SWITCH instruction, which may need to route tokens using one routing list for a true condition and another for a false condition. There are no examples of this in matrix multiply, but it is common in other programs.

```
MM-0
  #  opcode           #args args (value, port)            dest [i, p] route (t, f) {c}
================================================================================
  0  PROMPT           0                                -> [12,0] {B} *
  1  PROMPT           0                                -> [13,0] [7,0] [2,0] [5,0] [4,0] [3,0] {A} *
  2  UPPER_BOUND      2 (0.00,1)                       -> [6,2] [9,1] [11,0] *
  3  LOWER_BOUND      2 (1.00,1)                       -> [6,3] [14,0] *
  4  UPPER_BOUND      2 (1.00,1)                       -> [6,4] [15,0] *
  5  LOWER_BOUND      2 (0.00,1)                       -> [6,1] [16,0] *
  6  ALLOCATE         5 (2.00,0)                       -> [8,0] *
  7  LOWER_BOUND      2 (0.00,1)                       -> [9,0] [10,1] *
  8  FORKJUMP         2 (1.00,1)                       -> [17,0] *
  9  LE               2 (STKY,1)                       -> [10,0] *
 10  SWITCH           5 (1.00,2) (11.00,3) (2.00,4)    -> [18,0] [19,0] [21,0] {I} *
 11  DIST_LOPERATOR   1 (STKY,0)                       -> (12,0) *
 12  DIST_LOPERATOR   1 (STKY,0)                       -> (14,0) *
 13  DIST_LOPERATOR   1 (STKY,0)                       -> (15,0) *
 14  DIST_LOPERATOR   1 (STKY,0)                       -> (10,0) *
 15  DIST_LOPERATOR   1 (STKY,0)                       -> (11,0) *
 16  DIST_LOPERATOR   1 (STKY,0)                       -> (13,0) *
 17  DIST_LOPERATOR   1 (STKY,0)                       -> (16,0) *
 18  DIST_LOPERATOR   1                                -> (1,0) *
 19  PLUS             2 (1.00,1)                       -> [20,0] {NEXT-I} *
 20  D                2 (-11.00,1)                     -> [9,0] [10,1] {I} *
 21  DINV             1                                -> {SIGNAL} *
```

In SP MM-0 the PROMPT instructions acquire the A and B matrices used in the matrix multiply. The UPPER_BOUND and LOWER_BOUND instructions access the matrix headers to initialize the loop boundaries. ALLOCATE then remotely distributes the C array and feeds a FORKJUMP operator. This FORKJUMP is necessary for the array manager to have a place to return the array identifier it just allocated. The LE, SWITCH, PLUS, D, and DINV are the standard dataflow operators. The new PODS operator is the DIST_LOPERATOR, which performs the standard LOOP operator dataflow operations, but also sends its tokens to *all* PEs. This is how *i* is distributed.

In SP MM-1 below, there is the local equivalent of the DIST_LOPERATOR, the LOCAL_LOPERATOR, which sends its tokens only to itself. LOCAL_LOPERATORs are only used where the operations have already been distributed, and more distribution would simply add to the network overhead without generating more parallelism. MM-1 also has

the code for the range generator inserted into it, from instruction 0 to 18. The range generator is explained more fully in Appendix B.

```
MM-1
  0  INTERVAL_COUNT    1 (STKY,0)                              -> [1,1] *
  1  LT                2 (0.00,0) (STKY,1)                     -> [2,0] *
  2  SWITCH            5 (0.00,1) (1.00,2) (29.00,3) (3.00,4)-> [3,0] [5,0] [8,1] [31,0] *
  3  B_RANGE           3 (STKY,1) (0.00,2)                     -> [4,1] *
  4  GE                2 (STKY,0)                              -> [7,0] *
  5  E_RANGE           3 (STKY,1) (0.00,2)                     -> [6,0] *
  6  GE                2 (STKY,1)                              -> [7,1] *
  7  AND               2                                       -> [8,0] *
  8  SWITCH            5 (1.00,2) (9.00,3) (3.00,4)            -> [9,0] [10,0] [16,1] [17,0] *
  9  E_RANGE           3 (STKY,1) (1.00,2)                     -> [11,1] *
 10  B_RANGE           3 (STKY,1) (1.00,2)                     -> [11,0] [12,1] *
 11  LE                2 (STKY,1)                              -> [12,0] [16,0] *
 12  SWITCH            5 (1.00,2) (4.00,3) (3.00,4)            -> [13,1] [15,0] [19,1] *
 13  LE                2 (STKY,0)                              -> [14,0] *
 14  SWITCH            5 (STKY,1) (1.00,2) (-3.00,3) (0.00,4)-> [11,0] [12,1] *
 15  LE                2 (STKY,1)                              -> [16,0] [19,0] *
 16  SWITCH            5 (STKY,0) (STKY,1) (3.00,2) (1.00,3) (0.00,4)-> [17,0] *
 17  PLUS              2 (1.00,1)                              -> [18,0] *
 18  FORKJUMP          2 (-17.00,1)                            -> [1,0] [2,1] *
 19  SWITCH            5 (1.00,2) (12.00,3) (3.00,4)           -> [20,0] [26,0] [27,3] [31,0] {J} *
 20  LOCAL_LOPERATOR   1                                       -> (7,0) *
 21  LOCAL_LOPERATOR   1 (STKY,0)                              -> (2,0) *
 22  LOCAL_LOPERATOR   1 (STKY,0)                              -> (3,0) *
 23  LOCAL_LOPERATOR   1 (STKY,0)                              -> (4,0) *
 24  LOCAL_LOPERATOR   1 (STKY,0)                              -> (5,0) *
 25  LOCAL_LOPERATOR   1 (STKY,0)                              -> (6,0) *
 26  PLUS              2 (1.00,1)                              -> [28,0] {NEXT-J} *
 27  WRITE_ARRAY       4 (STKY,1) (STKY,2)                     -> {SIGNAL} *
 28  D                 2 (-17.00,1)                            -> [11,0] [12,1] {J} *
 29  GE                2 (STKY,0)                              -> [30,0] *
 30  SWITCH            5 (0.00,1) (-11.00, 2) (-19.00, 3)-> [19,0] *
 31  DINV              1                                       -> {SIGNAL} *
```

SP MM-2 is a simple local loop which performs a reduction-like operation. There is a carried dependency in MM-2's loop that causes it to be run on one PE and not distributed. The LOCAL_LINV operator routes the sum (S) back to its parent SP which is on the same PE since it is a local operation. This route uses route list 9 which is loaded into every routing unit.

```
MM-2
  0  LE                2 (STKY,1)                              -> [2,0] [1,0] *
  1  SWITCH            5 (1.00,2) (1.00,3) (3.00,4)            -> [3,2] [5,1] [4,0] [10,0] {TRIGGER} *
  2  SWITCH            5 (0.00,1) (1.00,2) (8.00,3) (1.00,4)-> [7,0] [11,0] {S} *
  3  READ_ARRAY        3 (STKY,0) (STKY,1)                     -> [6,0] *
  4  PLUS              2 (1.00,1)                              -> [8,0] {NEXT-K} *
  5  READ_ARRAY        3 (STKY,0) (STKY,2)                     -> [6,1] *
  6  MULT              2                                       -> [7,1] *
  7  PLUS              2                                       -> [9,0] {NEXT-S} *
  8  D                 2 (1.00,1)                              -> [0,0] [1,1] {K} *
  9  D                 2 (-9.00,1)                             -> [2,1] {S} *
 10  DINV              1                                       -> *
 11  DINV              1                                       -> [12,0] *
 12  LOCAL_LINV        1                                       -> (9,0) *
```

The routing file below is the "program" that the routing unit follows for sending tokens to different SPs. Notice that route list 9, used by MM-2, sends the sum to MM-1,

instruction 27, port 0. Checking MM-1 we see that instruction 27 is the WRITE_ARRAY instruction which is filling array C.

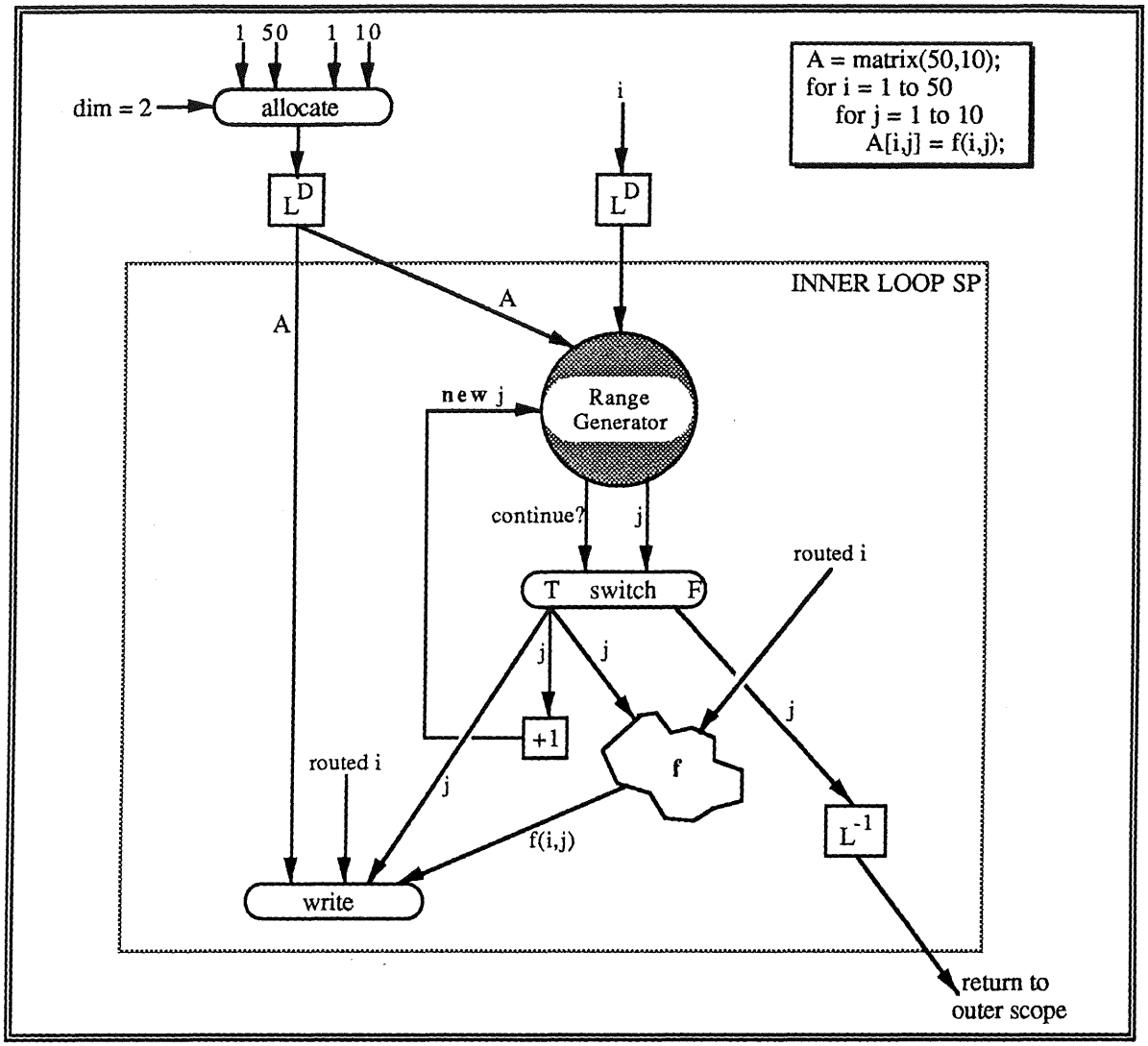
```
DISPLAYING ROUTES
  #     destinations [sp, inst, port]
=============================================================================
  1  -> [1, 25, 0] [1, 27, 2] [1, 4, 0] [1, 6, 1]
  2  -> [2, 0, 0] [2, 1, 1]
  3  -> [2, 0, 1]
  4  -> [2, 5, 0]
  5  -> [2, 3, 0]
  6  -> [2, 3, 1]
  7  -> [2, 5, 2]
  9  -> [1, 27, 0]
 10  -> [1, 13, 0] [1, 14, 1]
 11  -> [1, 15, 1] [1, 29, 0]
 12  -> [1, 22, 0]
 13  -> [1, 21, 0]
 14  -> [1, 23, 0]
 15  -> [1, 24, 0]
 16  -> [1, 27, 1] [1, 0, 0] [1, 3, 1] [1, 5, 1] [1, 9, 1] [1, 10, 1]
```

Appendix B: Range Generators

In this appendix, we explain the concept of range generators in detail. While it is not necessary for all to read this section, many readers may find it useful to understand how each PE restricts loop execution to its own portion of an array.

B.1. Objective and Usage

The objective of the range generator construct is to control which iterations of a distributed loop are to be executed by a given PE. A dataflow diagram which uses a 2-dimensional range generator is shown below. The range generator takes in the array [A], the outer index value [i], and the current index [j]. From these it produces the next index for which this PE is responsible. The dataflow diagram below is a graphical representation of the simple filling code in the upper right hand corner.



B.2. Boundary Table

Boundary tables are generated at allocation time and referenced by the range generator to determine the boundaries of its area of responsibility. In the figure below two types of boundary tables are shown, one for interleaved ranges and another for grouped ranges. In our experiments, grouped ranges were used because they generate fewer superpage boundaries. The example shown here is for PE #1 allocating a 6x300 array. The values *b1* and *b2* are the beginning values for a given range interval in each of the two dimensions; similarly for *e1* and *e2*. For different numbers of PEs (four in this example) different distributions are produced. The page size comes into play because pages are used

in caching and remote accesses. In this example the page size of 100 happens to split the interleaved ranges up into many intervals.

Interleaved Row

range_id	b1	e1	b2	e2
0	1	1	1	100
1	2	2	100	200
2	3	3	200	300
3	5	5	1	100
4	6	6	100	200

Grouped Row

range_id	b1	e1	b2	e2
0	1	1	1	300
1	2	2	1	200

Dim = 2, Size = 6x300, #PEs = 4, Page Size = 100, PE 1

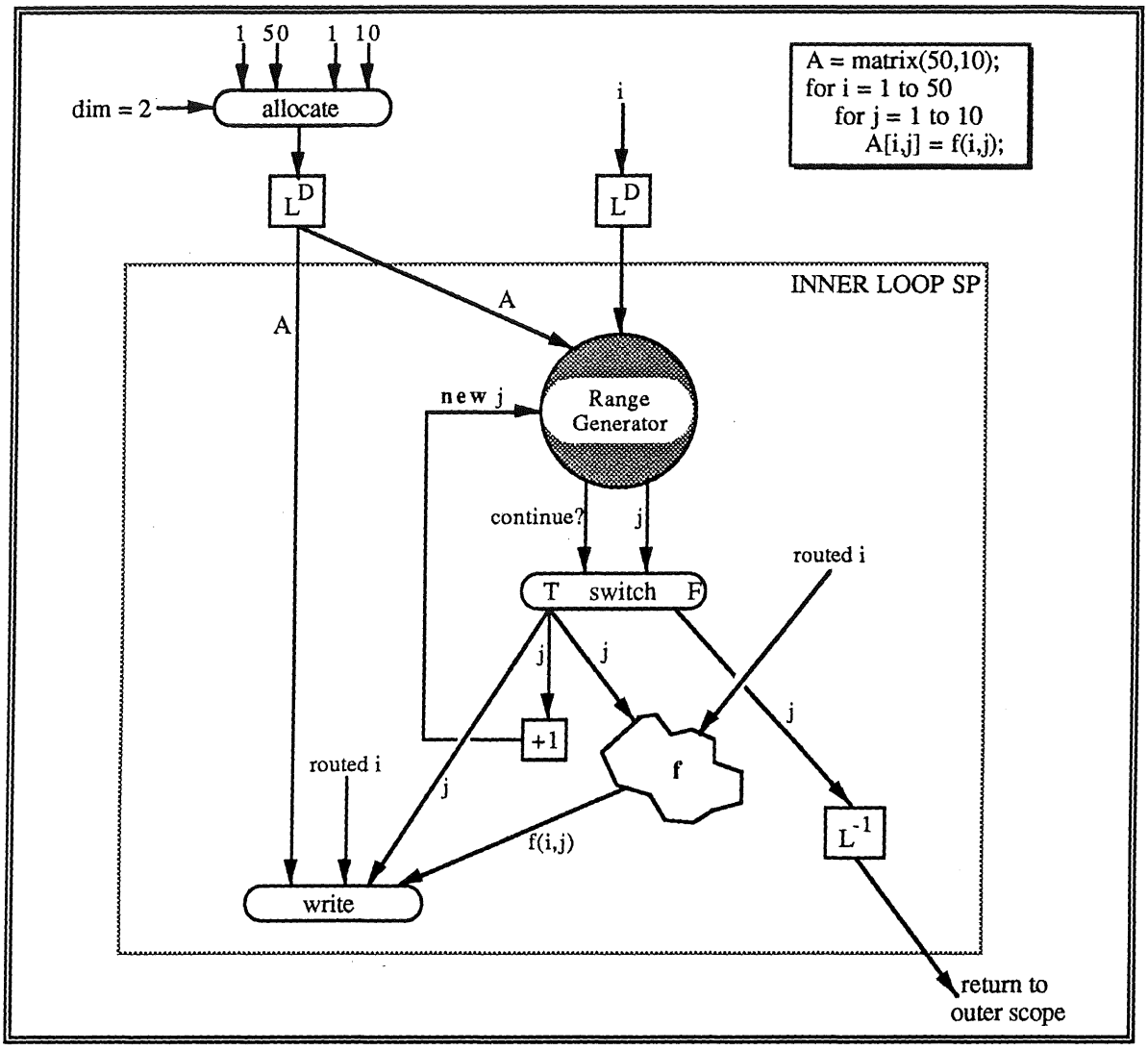
B.3. Algorithm

The algorithm for the range generator is fairly straight forward. It is important to note, however, that the general algorithm is parameterized. The algorithm shown here is for a simple ascending loop with a stepsize of one. For different stepsizes or directions, the range generator algorithm must be modified. However, the selection of algorithm can generally be done at compile time, so no more runtime overhead is used than necessary.

The general algorithm functions by repeatedly extracting ranges from the array boundary table. While within the range, the generator produces indices for elements within that range. The generator also keeps the loop alive by sending a continue token to the loop switch until all ranges have been exhausted.

Consider the boundary table from the grouped example above. If the index for the first dimension is 1 then the values 1-300 should be generated for the second dimension index. If the index is 2, then the values 1-200 should be generated for the second dimension index.

```
1    set N to interval count
2    set I to 0
3    if I > N then goto 14
4    read interval I
```



B.2. Boundary Table

Boundary tables are generated at allocation time and referenced by the range generator to determine the boundaries of its area of responsibility. In the figure below two types of boundary tables are shown, one for interleaved ranges and another for grouped ranges. In our experiments, grouped ranges were used because they generate fewer superpage boundaries. The example shown here is for PE #1 allocating a 6x300 array. The values *b1* and *b2* are the beginning values for a given range interval in each of the two dimensions; similarly for *e1* and *e2*. For different numbers of PEs (four in this example) different distributions are produced. The page size comes into play because pages are used

in caching and remote accesses. In this example the page size of 100 happens to split the interleaved ranges up into many intervals.

Interleaved Row

range_id	b1	e1	b2	e2
0	1	1	1	100
1	2	2	100	200
2	3	3	200	300
3	5	5	1	100
4	6	6	100	200

Grouped Row

range_id	b1	e1	b2	e2
0	1	1	1	300
1	2	2	1	200

Dim = 2, Size = 6x300, #PEs = 4, Page Size = 100, PE 1

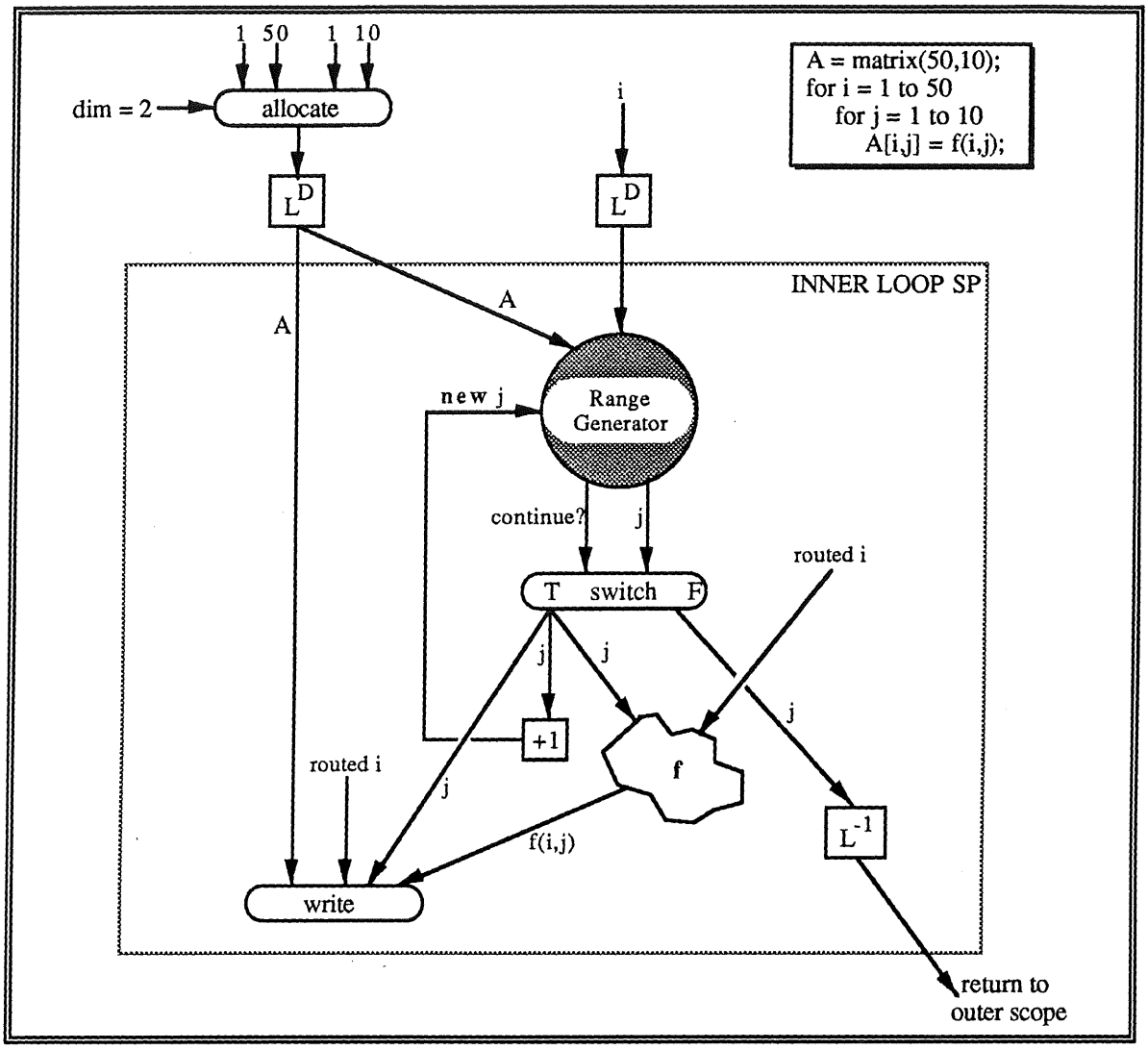
B.3. Algorithm

The algorithm for the range generator is fairly straight forward. It is important to note, however, that the general algorithm is parameterized. The algorithm shown here is for a simple ascending loop with a stepsize of one. For different stepsizes or directions, the range generator algorithm must be modified. However, the selection of algorithm can generally be done at compile time, so no more runtime overhead is used than necessary.

The general algorithm functions by repeatedly extracting ranges from the array boundary table. While within the range, the generator produces indices for elements within that range. The generator also keeps the loop alive by sending a continue token to the loop switch until all ranges have been exhausted.

Consider the boundary table from the grouped example above. If the index for the first dimension is 1 then the values 1-300 should be generated for the second dimension index. If the index is 2, then the values 1-200 should be generated for the second dimension index.

```
1    set N to interval count
2    set I to 0
3    if I > N then goto 14
4    read interval I
```



B.2. Boundary Table

Boundary tables are generated at allocation time and referenced by the range generator to determine the boundaries of its area of responsibility. In the figure below two types of boundary tables are shown, one for interleaved ranges and another for grouped ranges. In our experiments, grouped ranges were used because they generate fewer superpage boundaries. The example shown here is for PE #1 allocating a 6x300 array. The values *b1* and *b2* are the beginning values for a given range interval in each of the two dimensions; similarly for *e1* and *e2*. For different numbers of PEs (four in this example) different distributions are produced. The page size comes into play because pages are used

in caching and remote accesses. In this example the page size of 100 happens to split the interleaved ranges up into many intervals.

Interleaved Row

range_id	b1	e1	b2	e2
0	1	1	1	100
1	2	2	100	200
2	3	3	200	300
3	5	5	1	100
4	6	6	100	200

Grouped Row

range_id	b1	e1	b2	e2
0	1	1	1	300
1	2	2	1	200

Dim = 2, Size = 6x300, #PEs = 4, Page Size = 100, PE 1

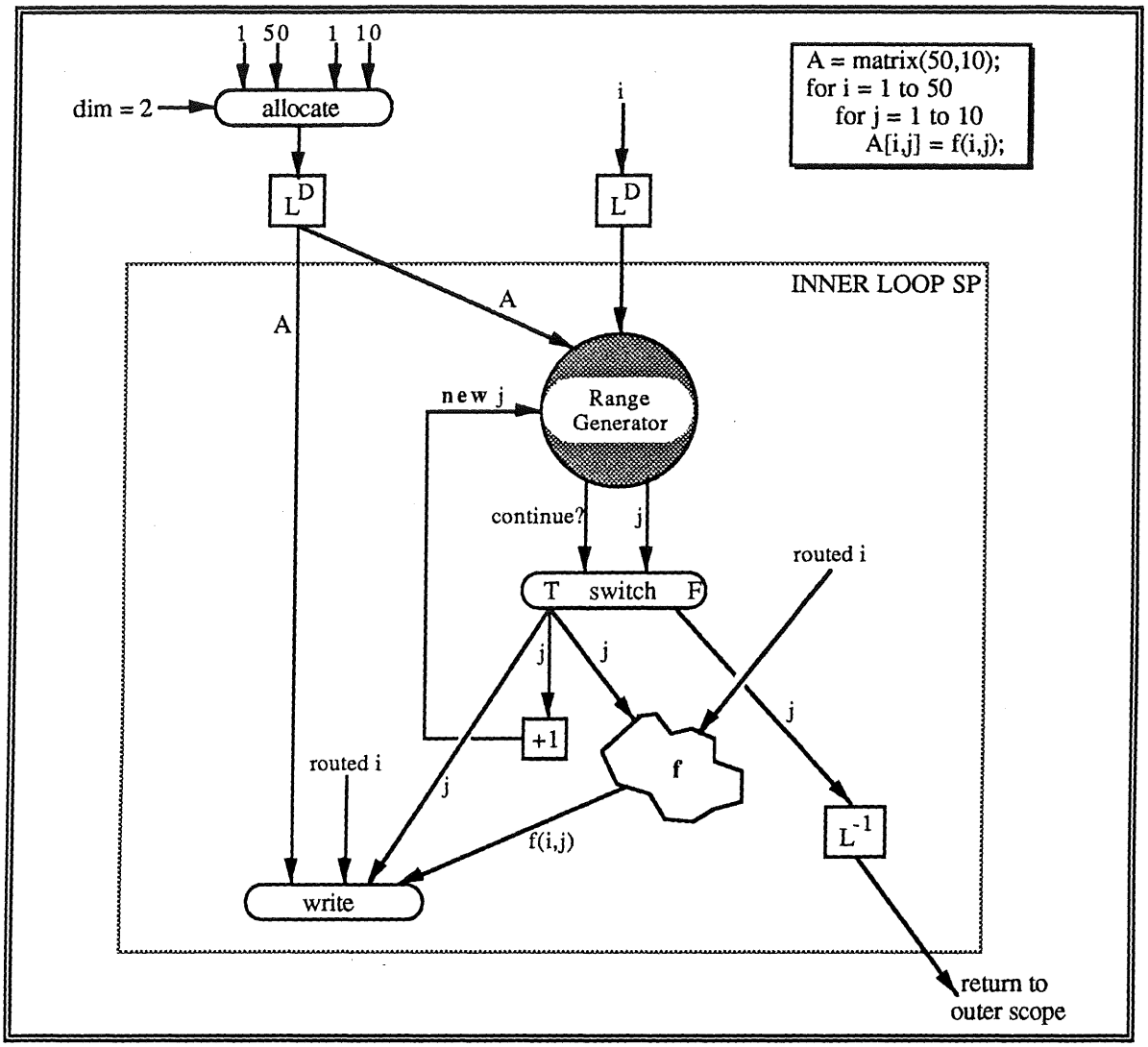
B.3. Algorithm

The algorithm for the range generator is fairly straight forward. It is important to note, however, that the general algorithm is parameterized. The algorithm shown here is for a simple ascending loop with a stepsize of one. For different stepsizes or directions, the range generator algorithm must be modified. However, the selection of algorithm can generally be done at compile time, so no more runtime overhead is used than necessary.

The general algorithm functions by repeatedly extracting ranges from the array boundary table. While within the range, the generator produces indices for elements within that range. The generator also keeps the loop alive by sending a continue token to the loop switch until all ranges have been exhausted.

Consider the boundary table from the grouped example above. If the index for the first dimension is 1 then the values 1-300 should be generated for the second dimension index. If the index is 2, then the values 1-200 should be generated for the second dimension index.

```
1    set N to interval count
2    set I to 0
3    if I > N then goto 14
4    read interval I
```



B.2. Boundary Table

Boundary tables are generated at allocation time and referenced by the range generator to determine the boundaries of its area of responsibility. In the figure below two types of boundary tables are shown, one for interleaved ranges and another for grouped ranges. In our experiments, grouped ranges were used because they generate fewer superpage boundaries. The example shown here is for PE #1 allocating a 6x300 array. The values *b1* and *b2* are the beginning values for a given range interval in each of the two dimensions; similarly for *e1* and *e2*. For different numbers of PEs (four in this example) different distributions are produced. The page size comes into play because pages are used

in caching and remote accesses. In this example the page size of 100 happens to split the interleaved ranges up into many intervals.

Interleaved Row

range_id	b1	e1	b2	e2
0	1	1	1	100
1	2	2	100	200
2	3	3	200	300
3	5	5	1	100
4	6	6	100	200

Grouped Row

range_id	b1	e1	b2	e2
0	1	1	1	300
1	2	2	1	200

Dim = 2, Size = 6x300, #PEs = 4, Page Size = 100, PE 1

B.3. Algorithm

The algorithm for the range generator is fairly straight forward. It is important to note, however, that the general algorithm is parameterized. The algorithm shown here is for a simple ascending loop with a stepsize of one. For different stepsizes or directions, the range generator algorithm must be modified. However, the selection of algorithm can generally be done at compile time, so no more runtime overhead is used than necessary.

The general algorithm functions by repeatedly extracting ranges from the array boundary table. While within the range, the generator produces indices for elements within that range. The generator also keeps the loop alive by sending a continue token to the loop switch until all ranges have been exhausted.

Consider the boundary table from the grouped example above. If the index for the first dimension is 1 then the values 1-300 should be generated for the second dimension index. If the index is 2, then the values 1-200 should be generated for the second dimension index.

```
1    set N to interval count
2    set I to 0
3    if I > N then goto 14
4    read interval I
```

5	increment I
6	if index 1 is not in interval I then goto 3
7	set J to start of interval I
8	if J is not in interval I then goto 3
9	if J is within loop bounds then set continue to TRUE else set continue to FALSE
10	send J to loop body
11	loop body
12	if new_J is within loop bounds then set continue to TRUE else set continue to FALSE
13	if continue then goto 11 else goto 8 (with J set to new_J)
14	(continue with program)

Algorithm for 2D Ascending Stepsize 1 Range Generator

In matrix multiply, the middle loop (MM-1) has the range generator code. The three new required operators are: INTERVAL_COUNT (retrieve the number of range intervals for this array); and B_RANGE / E_RANGE (retrieve the beginning and ending values for the specified range interval). These two new operators simply read entries from the array header (generated at allocation time). The additional SWITCHs and tests are necessary to prevent different PEs from generating the same indices. With range generators, each PE has the same code; only the local boundary tables are different.