REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Achieving Speedups for a Shared Memory Model Language on an SIMD Parallel Computer		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER 87-09-03
7. AUTHOR(s) Ray Greenlaw and Larry Snyder		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0264
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 September 1987 13. NUMBER OF PAGES 20
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) APL, mesh connected computer, performance, parallel processing, sequential algorithms		

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

The potential for speeding up a shared memory model sequential programming language, APL, by using an idealized non-shared memory parallel computer is investigated. We simulated the running of APL's dyadic, reduction, and subscript operators on a 4-connected mesh SIMD parallel computer. The simulation results indicate that these operations can be sped up significantly using parallelism. These findings support the thesis that parallelism can speedup a majority of "typical" APL programs and not just those programs that are especially suited to parallelization. The sequential APL language requires no sophisticated compila-

tion techniques like Paraphrase and PFC, because the operators in APL are inherently parallel. We fell that exploiting this property of APL is interesting.

Achieving Speedups for a Shared Memory Model Language on an SIMD Parallel Computer

Ray Greenlaw and Larry Snyder

Department of Computer Science
University of Washington

Technical Report 87-09-03
September 1987

(APL programming language)

Abstract

The potential for speeding up a shared memory model sequential programming language, APL, by using an idealized non-shared memory parallel computer is investigated. The authors ~~We~~ simulated the running of APL's dyadic, reduction, and subscript operators on a 4-connected mesh SIMD parallel computer. The simulation results indicate that these operations can be sped up significantly using parallelism. These findings support the thesis that parallelism can speedup a majority of "typical" APL programs and not just those programs that are especially suited to parallelization. The sequential APL language requires no sophisticated compilation techniques like Paraphase and PFC, because the operators in APL are inherently parallel. It is felt ~~We feel~~ that exploiting this property of APL is interesting. Keywords:

[illegible] processing, [illegible] parallel algorithms. [illegible]

This research has been funded by the Office of Naval Research Contract No. N00014-86-K-0264 and the National Science Foundation Grant No. DCR 8416878.

Authors' address: Department of Computer Science FR-35, University of Washington, Seattle, WA 98195.

1 Introduction

The main objective of this research is to determine how much speedup can be achieved by implementing a sequential shared memory model programming language on a non-shared memory model parallel computer with P processors. The sequential language is APL- A Programming Language, originally defined by Iverson [Ive62]. Many APL operators have arrays as data and exhibit a great deal of inherent parallelism. Budd [Bud84] showed an APL compiler designed for a vector processor in which most operations are performed in parallel. The inherent parallelism in APL is a central motivation for its choice. Our goal is to obtain speedups for any "typical" APL program and not just those especially suited to parallelization. Therefore, to simplify our simulation it is sufficient to focus on APL operators that studies have shown comprise the majority of average APL programs. One such study is described by Saal and Weis [SW75].

In fact, the 80-20 rule, which says that 80% of the operations used come from a subset consisting of only 20% of the total operators, was observed to hold for APL [SW75]. APL primitive scalar functions, the reduction operator, the subscript operator and user function calls were found to comprise the majority of a wide variety of APL programs. We simulated the first three of these operations on a mesh connected computer [Bea68]. The findings indicate that appreciable speedups can be obtained for all three of these operations. These results suggest that a large portion of an average APL program can be sped up significantly on a mesh connected computer. In addition, if the remaining part of the APL program could be run without degrading the performance, then the overall running time of the program could be decreased by a significant percentage.

Part of our concern with this research is to make existing sequential programs more efficient. In addition, we are concerned with utilizing parallel machines in an effective manner. Parallel computers are currently being manufactured and some of the difficulties in programming these machines have been investigated. In Nelson's thesis [Nel87] several parallel programming paradigms are identified. Although these paradigms are not conceptually much more complicated than sequential programming paradigms, actual machine specific implementations applying these techniques to particular problems are complex. The need for additional tools to aid in the design of parallel programs has been recognized and there are several existing parallel programming environments, such as Poker [Sny84] and Simple Simon [CBHH87], that provide some of these tools.

However, parallel programming is not yet as well understood as sequential programming. Our research indicates that existing representative APL programs could be compiled to run more efficiently on existing parallel machines without having to reprogram them into parallel solutions. Assuming the existence of such a compiler, new or existing sequential programs could be compiled and run on the parallel machine to see if the speedups obtained are near optimal. If the converted sequential program ran efficiently, then it would not need to be reprogrammed for the parallel machine.

Although this motivation, to run existing sequential programs in parallel and to simplify writing new parallel programs through the use of a sequential language, is similar to the work of Kuck [KKLW80] and Kennedy [AKPW83] in FORTRAN restructuring, the details differ in one significant way: no sophisticated compiler techniques are required to recognize the parallelism we exploit in APL. The parallelism is all found *within* the semantics of a single operator; to exploit this parallelism we simply substitute a parallel implementation of the operator for the sequential one. Since scalar operators are elementwise independent over all elements of an array, their parallel speedup should be expected to be near linear (e.g. summation). Since the reduction operator is an accumlation operator, its parallel implementation requires combining all elements, which on the mesh should yield a speedup factor on the order of the mesh dimension. A parallel implementation of general subscripting might be expected to yield speedups on the order of the number of items in the subscript vector stored per processor times the dimension of the mesh.

The remainder of the paper is outlined as follows. In Section 2 the possible speedups obtainable for a hypothetical APL program are analyzed. We assume the program has certain operator usage percentages and make assumptions about the parallel speedups for these operators. Our case analysis using Amdahl's Law [Amd] suggests caution at being overly optimistic about the potential speedups obtainable for complete programs unless all segments of the program parallelize well. In Section 3 the assumptions used in conducting our simulation, the algorithms used for implementing the primitive scalar, reduction, and subscript operators, and the general methodology of our simulation are described. In Section 4 the timings we obtained from the simulation are presented. In Section 5 we interpret our data and compare it to data obtained for a VAX running APL and in Section 6 conclusions and further research questions are presented.

2 Analysis of Possible Speedups Using Parallelism

In this section we examine some of the possible speedups attainable in implementing APL on a non-shared memory parallel machine. For a non-shared memory model parallel computer one of the fundamental problems is how to communicate data efficiently between processors. Local communication is usually not too expensive, but if our APL implementation requires a great deal of arbitrary communication, then the implementation may become too inefficient to be useful. Many APL operators have arguments that are arrays and under certain data allocation schemes they adapt well to parallelism [Sch87]. For example, since the arguments to the dyadic operators must conform in size, it is easy to assign to each processor data elements that are to be operated on pairwise. In such cases, theoretically, near optimal speedup over the sequential case are obtainable because there is no need to perform any global communication. In general, for operations like subscripting that require arbitrary communication, we can not hope for such complete speedups.

In what follows we conduct a three case analysis using Amdhal's Law [Amd] in order to examine the possible speedups that might be achieved under various implementations of APL on a non-shared memory parallel machine. We fix the percentage of time required for several important APL operators and vary assumptions about the amount of speedup achievable for each these operators in a parallel implementation. No specific non-shared memory machine model is assumed but just that the parallel machine has P processors.

In [SW75] the study conducted on APL programs indicated the operator usage percentages depicted in Table 1. Table 2 shows the assumptions made about the amount of speedups achievable using P processors. In all three cases presented we assume that using P processors gives an optimal speedup of P for the scalar primitive operators. We will vary the assumptions about the subscripting operations as follows: in case 1 we assume that a slowdown by a factor of 2 occurs, in case 2 we assume no speedup, and in case 3 we suppose a speedup by a factor of $log\ P$. We assume that all remaining operations will require about the same amount of time in cases 1 and 2, and that in case 3 a speedup by a factor of $log\ P$ is achieved.

Suppose a "typical" APL program requires time T to execute on a serial machine, what do the operator usage percentages in Table 1 and the assumptions in Table 2 imply about the parallel implementation's running time for the same program?

Operation	Percentage of Use [SW75]
scalar primitives	73%
subscripting	18%
reduction	2.6%
all others combined	6.4%

Table 1: APL operation usage percentages.

Operation	Case 1	Case 2	Case 3
scalar primitives	P	P	P
subscripting	.5	1	log P
all others combined	1	1	log P

Table 2: Speedup factors assumed for case analysis using P processors.

1. **CASE 1** For the portion of the execution time attributed to the scalar primitives, a speedup of P is achieved. For the portion of the program due to the subscript operation, a slowdown of by a factor of two occurs. The last part of the program's execution time, which we assume is due to all other operations, remains the same. Therefore, the following formula describes the overall run time of the parallel implementation:

$$\frac{.73T}{P} + .45T \tag{1}$$

This shows that as the number of processors P gets very large the running time of the program approaches $.45T$. Thus, asymptotically the best overall speedup possible is about a factor of two.

2. **CASE 2** For the portion of the execution time attributed to scalar primitives, a speedup by a factor of P is achieved. For the remaining part of the program there is no change. Therefore, the following equation describes the overall run time:

$$\frac{.73T}{P} + .27T \tag{2}$$

This shows that as the number of processors grows large the running time of the program approaches $.27T$. Thus, at best a factor of four speedup is obtainable. Notice, this is only slightly better than the speedup achieved in case 1.

3. **CASE 3** For the portion of the program due to scalar primitives a speedup of P is obtained. For the remaining part of the program a speedup of *log P* is achieved. This is due to the more efficient implementations of the subscript operator and all other operators. The following equation describes the overall run time:

Number of PEs	Case 1	Case 2	Case 3
100	75.5	50.5	25.5
1,000	75.05	50.05	16.71
10,000	75.005	50.005	12.255
100,000	75.0005	50.0005	10.0005

Table 3: Parallel running times for a program, which took 100 seconds sequentially, assuming the equations presented for the three cases(logs are base 10).

$$\frac{.73T}{P} + \frac{.27T}{log\ P} \tag{3}$$

Equation 3 shows that with additional processors we continue to get marginal increases in speed. Note, that the assumptions in case 3 implied more than a constant factor speedup for all "parts" of the program, and therefore; more than a constant factor speedup for the overall program was obtained.

Table 3 depicts the parallel running times for cases 1 through 3 on an APL program assumed to have taken 100 seconds to run sequentially. Notice that for cases 1 and 2 the speedups achieved by adding another order of magnitude of processors are almost negligible. In case 1 a speedup by a factor of about four thirds is obtained and in case 2 a speedup by a factor of two. In case 3 notice that additional processors help reduce the running time of the program significantly. In addition, for case 3 adding more processors achieves more than a constant factor speedup.

The conclusions that we draw below are based on "average" APL programs. Obviously, there is no one program that is representative of all APL programs, however, by an average APL program we mean one that is as representative of APL programs as possible. Some APL programs may be particularly suited to a parallel implementation and others may not be parallelizable at all. The conclusions are also based on the assumptions given in Table 2. Clearly, these are simplifying assumptions but they illustrate the limiting cases.

Our analysis illustrates the point that all significant parts of typical programs need to be sped up if an appreciable overall speedup is to be achieved. In particular, if one significant portion of a program can not be sped up at all using parallelism, then this will serve as a lower bound on the possible speedup for the entire program. For example, if there is no hope of speeding up 20% of a program, then the parallel implementation's run time can be at best a factor of 5 better than the serial implementation's regardless of the number of processors that are used. This suggests that we bias our implementation towards the operation for which it is most difficult to achieve speedups even if it is at the expense of less efficient implementations for the remaining operators.

The primitive scalar operations of APL seem to adapt well to parallelism as do some of the other operators listed with them. This is, of course, assuming a parallel implementation in which global communication is kept at a minimum. However, the subscripting operations seem more difficult to parallelize due to communication costs. There is no economical way of storing the data to be operated on so that all possible subscripting operations can be sped up on a non-shared memory parallel computer. This is because the subscript operator can require any permutation of the data be returned. In the next section we focus on how to implement the APL subscript

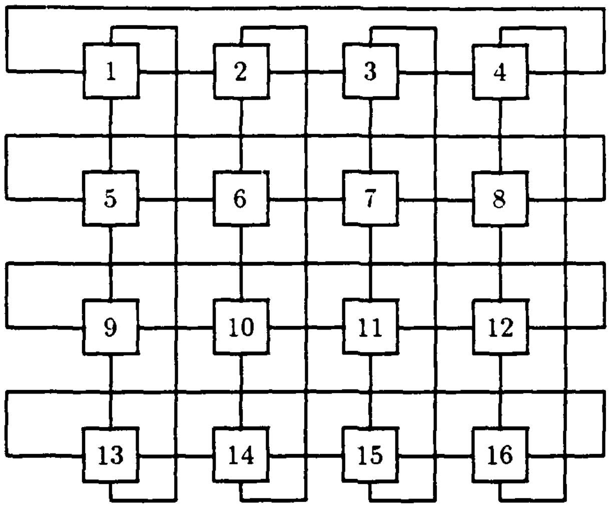

Figure 1: 4 × 4, 4-connected SIMD mesh (control PE is not shown).

operator efficiently on a non-shared memory model in a way that still allows some speedups for the other APL operators. If we succeed at this task for typical programs, then we can claim that speedups for average APL programs using parallelism are achievable.

3 A Simulation of Several Major APL Operators on a Mesh

The non-shared memory parallel computer used is the 4-connected SIMD (single instruction stream, multiple data stream) mesh [Bea68] with toroidal connections and a control processor. The interconnection structure for a 4 × 4 mesh is shown in Figure 1. Each processor is numbered according to its position in the mesh and the control processor (not shown) is numbered 0.

The machine is composed of the control processor that broadcasts instructions to the processors in the mesh. The control processor can broadcast an instruction to all PEs in the mesh in unit time. Although this is somewhat unrealistic assumption for a large mesh, it is acceptable because operations requiring arbitrary communication on an $n \times n$ mesh will require time at least $O(n)$ anyway. Since the control processor broadcasts instructions, there is a single instruction stream. Each processor can either execute the instruction on its own data or wait for that step. Thus, there is a multiple data stream. Each processor has its own local symbol table. Thus, there is non-shared memory. The control processor has in its symbol table all scalar values. All APL arrays are stored in the mesh in row major order; thus, multidimensional arrays are "strung out" in a convenient way so that they can be treated similarly to one dimensional arrays. For the most part, we assume that we are dealing with one dimensional arrays although the results generalize to arrays of higher dimensions. Schaad has considered other allocations [Sch87].

For an $n \times n$ mesh, communication between any two PEs (processing elements or processors) can be done in time $O(n)$. This will serve as a lower bound on the amount of possible speedup achievable for specific instances of APL operations requiring arbitrary communication. Certain instances of subscripting are examples of such operations. Notice there are other bounded degree networks in which arbitrary communication requires less time. In general though, their interconnection structures and message routing algorithms are more complicated than the mesh's [Upf84]. Besides, if we can demonstrate effective speedups on the mesh then this provides us with strong evidence that such speedups are also achievable on some of these other networks.

In the next three subsections we focus on algorithms for implementing the APL dyadic, reduction, and subscript operators on the 4-connected mesh. For analyzing the implementation of the subscript operator it useful to isolate the computational aspects of the problem by considering a shared memory model implementation.

3.1 Dyadic Operator

We choose to implement the dyadic operator because the Saal and Weis study indicated the primitive scalar operations are frequently used and the dyadic operator is representative of this group of operators, moreover, the actual parallel implementation of the operator can serve as a benchmark of the types of speedups expected from our simulation. The simulation's running time for the dyadic operator can serve as a benchmark because theoretically it seems reasonable to hope for a near perfect speedup in implementing it for large data sizes and large mesh sizes. This is assuming that the elements of the pairwise operation can be stored in the same PE.

Given that we choose to store our data in row major order (see [Sch87] for a study conducted on the impacts of different storage allocation schemes and how they affect the parallel running times of APL programs implemented on the n-cube) the obvious efficient way to compute the dyadic operator is as follows:

Algorithm 3.1 *Dyadic Algorithm*

1. The control processor broadcasts to all PEs that the next instruction is dyadic and sends a message containing the destination, the operands and the operator type.

2. In parallel all PEs search for the operands in their local symbol tables and perform the required operation.

3. Step 2 is repeated as many times as necessary, when each PE has more than one data element.

4. The result is stored in the appropriate destination. Note, the destination of the result is always in the same PE where the result was computed in.

As an example on a 10 × 10 mesh, consider how the following APL dyadic instruction would be implemented; in the example A and B are initialized to the first 256 integers.

Example 3.1 $A \leftarrow \iota 256$
$B \leftarrow \iota 256$
$C \leftarrow A + B$

Initially the control processor broadcasts that the dyadic plus operation is to be performed on A and B with the result to be assigned to C. Each PE initially contains either 2 or 3 elements of A and the same number of elements of B. For example, PE 1 contains elements 1,101 and 201

Datasize	Vax 8500	100 PEs	256 PEs	1024 PEs
256	10	3	4	-
512	30	7	5	-
1024	50	15	11	2.6
1536	60	22	14	2.8
2048	80	31	17	3.1
2560	100	45	22	3.5
3072	120	57	30	4.0
3584	137	74	34	4.4
4096	150	87	41	5.0

Table 4: Timings in milliseconds for the APL dyadic operation.

of arrays A and B. In parallel all PEs find the corresponding pairs of elements of A and B they contain and perform the appropriate operation on these elements. Notice, that the result is also stored in the same PE and this adheres to the row major order storage convention for arrays. Thus, Algorithm 3.1 does not require any communication between PEs under the chosen data allocation scheme.

We analyze the expected speedup over the sequential case for the Dyadic Algorithm 3.1. Suppose the following dyadic instruction is to be executed:

Example 3.2 $C \leftarrow A + B$

where $|A| = |B| = l$. For the parallel case an $n \times n$ mesh is assumed. Sequentially the operation takes $O(l)$ time because each addition needs to be performed separately. In the parallel case there would be at most $\lceil \frac{l}{n^2} \rceil$ elements per PE, so the operation takes $O(\frac{l}{n^2})$ time. Theoretically then, we would expect a perfect speedup for this implementation. The simulation results obtained for Algorithm 3.1 are shown in Table 4. They support this analysis.

3.2 Reduction Operator

The reduction operator is an interesting one to implement because a certain amount of global communication is required. The following algorithm describes the way that we choose to implement the reduction operator.

Algorithm 3.2 *Reduction Algorithm*

1. The control processor broadcasts to all PEs that the next instruction is reduction and sends a message containing the destination, the operand and the operator type.

2. In parallel all PEs search for the occurrences of the operand in their local symbol tables and perform the required operation on all elements they contain. (For operations that are not associative and communative, e.g. subtraction and division, care must be taken concerning the order in which the operations are performed on the actual array elements).

3. These partial results are all forwarded to the leftmost column of the mesh.

4. The PEs in the leftmost column aggregate the partial results while the other PEs remain idle.

5. The results are sent to PE1 where they are "totaled" for the final solution to the reduction.

As an example on a 10 × 10 mesh, consider how the following APL reduction instruction would be implemented; in the example A is initialized to the first 256 integers.

Example 3.3 $A \leftarrow \iota 256$
$C \leftarrow +/A$

Initially the control processor broadcasts that the plus reduction operation is to be performed on A with the result to be assigned to C. Each PE initially contains either 2 or 3 elements of A. Each PE sums its elements. For example, PE1 sums elements 1,101 and 201 of A. Using general data movement techniques on the mesh [Ull84] these partial sums are then sent to the leftmost column of processors. In this case the ten results obtained for each row are summed up in the leftmost column. These ten new results are sent to PE1 where the final result of the reduction is computed. Notice, Algorithm 3.2 requires any given PE to receive at most one message at each step. We analyze the expected speedup over the sequential case for the Reduction Algorithm 3.2. Suppose the following reduction instruction is to be performed:

Example 3.4 $C \leftarrow +/A$

where $|A| = l$. For the parallel case an $n \times n$ mesh is assumed. Sequentially we would expect the operation to take $O(l)$ time because each addition needs to be performed separately. In the parallel case there would be at most $\lceil \frac{l}{n^2} \rceil$ elements stored in each PE. Therefore, the initial summing takes $O(\frac{l}{n^2})$ time. Step 3 in Algorithm 3.2 requires time $O(n)$ because n values need to be shifted to the leftmost column with one value coming from each PE. Step 4 requires time $O(n)$ because the PEs in the leftmost column have $O(n)$ elements to sum up. Step 5 requires time $O(n)$ since $O(n)$ values must be sent to PE1 and then summed up in PE1. Therefore, the overall running time of Algorithm 3.2 is $O(n)$ when $l < n^3$ and $O(\frac{l}{n^2})$ for $l > n^3$. Therefore, when $l > n^3$ we get a perfect speedup asymptotically.

Note, in the analysis the constant implied by the O notation is small. Furthermore, we have two steps in which communication between PEs is required and this is independent of the number of elements per PE. In the analysis we have charged one unit of time for each communication step. This analysis shows that Algorithm 3.2 yields appreciable speed up over the sequential case whenever $n^3 < l$ and runs in about the same time when n and l differ by a constant. The simulation results obtained for Algorithm 3.2 are shown in Table 5.

3.3 Subscript Operator

APL has a more general subscript operator than most other programming languages [GR76]. For example, in most languages access is to only one element of an array at a time. The expression $A[i]$ is used to retrieve the i^{th} element of array A. The index i to A must have a scalar value. In APL, the index into an array can be a vector. For example, $A[V]$ retrieves the elements from A whose indices match those specified by the vector V. The only restrictions to the indices appearing in V is that they be in the "domain" of A, i.e. legitimate subscripts. Consider the following example:

Datasize	Vax 8500	100 PEs	256 PEs	1024 PEs
256	1	1.1	1.2	-
512	20	1.6	1.3	-
1024	30	2.0	1.4	3.3
1536	60	2.8	1.5	3.6
2048	80	4.6	2.1	3.3
2560	90	6.6	2.3	3.5
3072	100	8.2	2.8	3.3
3584	120	13.4	3.0	3.8
4096	140	15.0	3.8	3.4

Table 5: Timings in milliseconds for the APL reduction operation where local computation is performed first in each PE.

Example 3.5 $A \leftarrow$ '*abcde*'
$V \leftarrow 1\ 4\ 4$
$A[V]$

add

In this example, the domain of A is 1 ... 5. Notice, the index vector V can have repeat elements. We analyze the running time for the sequential case and assume no special optimizations are performed based on the contents of the index vector V. To perform the operation $A[V]$ we need to compute separately the addresses of all of those elements specified by V. Therefore, if there are m elements in V then time $O(m)$ is required to retrieve the corresponding elements from A. The question addressed for the remainder of Section 3.3 is whether or not this computation can be sped up significantly using P processors.

3.3.1 Shared Memory Implementation of the Subscript Operator

Our primary interest is to implement APL on a non-shared memory model parallel computer, however, we first focus on a *shared* memory model in order to isolate the computational aspects of the problem. An SIMD shared memory model with $P + 1$ processors is assumed. Either the SIMDAG model [Gol77] or the CREW-PRAM model [FW78] will suffice. We also assume the model has a control processor that broadcasts the type of APL instruction to be executed to the other processors. The P processors can read from a global memory in unit time.

Suppose the control processor's program has the "instruction" $A[V]$ as it's next instruction. We assume A has size n and V has size m and, A and V are both stored in global memory. If the length of V is small then not much parallelism can be utilized. Therefore, we assume $m \geq \log P$. Under these assumptions, there are several ways to exploit parallelism in the subscripting problem. In what follows we describe two algorithms in which we assume the control processor, when encountering a subscript operator, broadcasts the name of the array to be subscripted and the name of the array to use as a subscript. The length of an array is stored in global memory in a global symbol table.

Algorithm 3.3 *Shared Memory Subscript Algorithm A*

1. In parallel for all i, processor P_i uses $V[i]$ as an index into A to obtain the $V[i]^{th}$ element of A.
2. In parallel for all i, P_i stores this result in the appropriate location.
3. In parallel for all i, P_i repeats the above steps k times, where k is the smallest integer such that $\frac{|V|}{P} < k + 1$.

Algorithm 3.4 *Shared Memory Subscript Algorithm B*

1. In parallel for all i, processor P_i computes $k = \lceil \frac{|V|}{P} \rceil$ to determine how many elements from A it will obtain.
2. In parallel for all i, P_i obtains elements $V[ik + i] \ldots V[i(k + 1) + i]$ of A.
3. In parallel for all i, P_i stores these results in the appropriate locations.

In Algorithm A processor P_i will probably not be accessing consecutive elements of A. However, in Algorithm B if the indices of V are consecutive, then P_i can save additional time by incrementing the address of the first element of A it computed by the size of an element of A, thus obtaining the address of the second element of A, and so on. Notice, if V contains duplicate entries then both algorithms may require concurrent reads.

How much speedup do these algorithms yield over the sequential case? Each of the P processors can access one element of A at each step. Therefore, since we assumed $m = |V| \geq log\,P$ a factor of at least $log\,P$ speedup is obtained. In cases where $|V| \geq P$ a factor of $O(P)$ speedup is achieved, which is optimal using P processors.

The following example illustrates the theoretical speedups Algorithms A and B provide. Consider the following sequence of APL instructions:

Example 3.6 $A \leftarrow \iota 1000$
$B \leftarrow A[2\,4\,6 \ldots 1000]$

Using a sequential computer, the second instruction would take roughly 500 steps. In the parallel case, assuming 100 processors, the second instruction would take about 10 steps using either of the shared memory algorithms described previously. In the next section our focus returns to the non-shared memory implementation of the subscript operator.

3.3.2 Non-Shared Memory Implementation of Subscript Operator

Let's assume the control processor broadcasts the subscript instruction in the same manner as before. Several problems arise in trying to convert either shared memory algorithm into an algorithm for the non-shared memory case. In particular, suppose processor P_i contains element $V[j]$. P_i may not contain element $A[V[j]]$ in which case large communication costs may be incurred in trying to obtain this value. In general, it is not possible to keep these values together since the subscripting operation pairs arbitrary sources and destinations; nevertheless, we may still be able to keep the communication costs low enough to benefit from parallelism.

Another problem that arises in converting Algorithms A and B to a non-shared memory model is that both algorithms use concurrent reads. This again increases the communication costs in the non-shared memory case. We contend that many uses of the subscript operator are to retrieve values from a set of data without duplicating any of these values. There are other operators that can be used for duplicating data items. For example, the APL replication operator [GR76] is frequently used to expand data. Therefore, in the remainder of this paper it is assumed that the indices in the index set V are distinct. In particular, this implies that the length of the index vector is less than or equal to the length of the vector. Of course, if only a constant number of duplicates occur in the index set, then our results would not be significantly affected.

The 4-connected mesh was chosen to implement the subscript operator on because of its simplicity, although other parallel models [Sto70] [PV81] would work at least as well. Suppose there are P processors in the mesh, then arbitrary PE to PE communication requires time $P^{.5}$. In fact, if all PE's are to send out messages and all messages have different destinations then to send out all P messages requires only time $O(P^{.5})$ using standard data flow techniques [Ul184]. Let's assume again that all arrays are stored in the mesh in row major order. Initially assume there is only one element of the array stored per PE. The following algorithm shows how to obtain a factor of $P^{.5}$ speedup for the APL subscript instruction $B \leftarrow A[V]$ when $|V| \geq P^{.5}$.

Algorithm 3.5 *Non-Shared Memory Subscript Algorithm C*

1. In parallel for all i, P_i sends $P_{V[i]}$ a message saying $A[V[i]]$ is the value of $B[i]$
2. In parallel for all i, $P_{V[i]}$ sends P_i the value of $A[V[i]]$ which P_i stores as $B[i]$.

Algorithm C is easily generalized to the case where each PE contains more than one element. Suppose the PEs have k elements apiece, then Algorithm C is repeated k times, once for each element. We call the modified version of Algorithm 3.5 Algorithm C$'$. Note, in Algorithm C$'$ we can no longer assume that the messages the PEs are sending out have distinct destinations. Thus, potentially there will be multiple messages being sent to the same PE during one communication phase. We discuss the expected and worst case behaviors of Algorithm C$'$.

We show that for suitable array sizes the "expected" speedup achieved by Algorithm C' is significant. Let $m = |V|$ and $k = \lceil \frac{m}{P} \rceil$. The sequential algorithm for performing the subscripting instruction $B \leftarrow A[V]$ requires $O(m)$ steps. We break down our analysis into separate cases for different values of k in the parallel case.

1. **CASE 1** where $k = O(1)$.
   Using a row major order data allocation scheme, each PE has only $O(1)$ elements. Since we assumed the elements of V were distinct, using Algorithm C$'$ to perform the subscripting requires time $O(P^{.5})$. This is based on the communication time required by the algorithm. Note, there can be at most a constant number of messages colliding at any given PE during one communication phase. The sequential algorithm requires $O(P)$ time since we assumed $m = O(kP)$, so a speedup by a factor of $O(P^{.5})$ is achieved.

2. **CASE 2** where k is no longer bounded by a constant.
   The following scenario describes the worst case behavior of Algorithm C$'$. Suppose $m = P^2$ and V is such that at each step all P processors want to access the data in the same PE. The communication that is required at each stage becomes $O(P)$. Therefore, the running time of

Datasize	Vax 8500	100 PEs	256 PEs	1024 PEs
256	10	9.2	6.8	-
512	40	20.9	14.5	-
1024	80	37.5	27.6	22.1
1536	109	61.5	44.1	46.1
2048	160	86.0	57.3	42.7
2560	210	121.1	70.0	68.2
3072	260	179.3	87.1	64.3
3584	290	205.3	102.6	98.5
4096	350	219.4	114.9	85.6

Table 6: Timings in milliseconds for the APL subscript operation

Algorithm C′ is $O(P^2)$. This is the same to within a constant factor as the running time in the sequential case. Therefore, in the worst case no speedup is achieved.

The number of collisions that occur during the communication phase of Algorithm C′ depends on the values in V and on how the data is allocated in the mesh. Case 2 shows that we cannot expect a speedup of a factor of $O(P^{.5})$ over the sequential algorithm by Algorithm C′ in all cases but we can probably expect more than a constant factor speedup in the average case. The simulation results are shown in Table 6. Our experimental data supports this theoretical analysis.

In review, the assumptions made about the subscripting operation were as follows:

1. The index vector V contained unique entries or at least the number of message collisions caused by duplicate entries during the communication phase of Algorithm C' was a constant.
2. When there were more than a constant number of data entries per PE, the values in V were such that the number of collisions occurring during the communication phase of Algorithm C′ was small.

If these two assumptions hold for an "average" APL program then theoretically we can expect to be able to achieve a significant speedup using parallelism for an average program. This is true because the remaining parts of an average program seem to adapt well to parallelism. In the next section details of the simulation are presented.

4 A Description of the Simulation

In the previous section described the algorithms we used in the simulation of the APL dyadic, reduction, and subscript operators on a mesh connected computer. We presented Algorithm 3.1 for the dyadic operator, Algorithm 3.2 for the reduction operator and a modified version of Algorithm 3.5 in section 3.3.2 for the subscript operator. In this section some simplifying assumptions used in the simulation are described and the manner in which the timings from our simulation comparing the APL operators implemented on a VAX 8500 to those implemented on the simulated mesh are discussed.

As mentioned previously, all APL arrays are assumed to be stored in row major order. We simulated the mesh computer by a sequential program written in the C programming language [KR78]. When we ran the simulations to obtain "parallel" timings, we took the total running time of the executing sequential program and divided it by the number of processors used for that particular simulation. For cases in which all processors were "active" at all steps, this estimate gives a good estimate of the actual parallel time. For cases in which some PEs were idle; of course, an overly optimistic parallel running time is computed.

All simulations were conducted on a VAX 8500. The version of APL we compared the simulation results to also ran on a VAX 8500 and was written in C. The version of APL used was originally written by K. Thompson at Bell Labs. The version we compared against was Purdue/EE's APL written in part by J. Bruner and A. Reeves. Since our simulations were performed on the VAX, this meant that the processors in the mesh were also VAXes. Of course, the processors need not have been full blown VAXes but for our purposes the PEs were assumed to be of the same speed as a VAX 8500 for the operations simulated.

One additional note about the simulation was that we did not assign an explicit cost for message passing. In the simulation a program's communication basically amounts to an assignment statement. This, of course, is unrealistic since communication cost in a large mesh would amount to more than this. However, if the actual message passing cost for the mesh is a small constant factor than the results are still valid. Since the communication patterns in the mesh are fairly predictable and regular, shifting the data curves slightly to the right by a constant factor would take into account this communication cost. In addition, for the dyadic operator there is no communication cost incurred so the results are valid; and furthermore, for the reduction operator there are only two "global" communication phases.

4.1 Dyadic Simulation

The timings obtained for the dyadic simulation are shown in Table 4. The results were plotted and the graph is shown in Figure 2. The following sequence of APL instructions was executed by the simulation program and by Purdue/EE APL.

Example 4.1 $A \leftarrow \iota x$
$B \leftarrow A$
$C \leftarrow A + B$

Timings were obtained for the third step. The values for x varied from 256 up to 4096. We varied the number of processors in the mesh from 100 up to 1024. Thus, for example in Table 4 the entry 31 corresponding to 2048 and 100 means that under our assumptions on a mesh with 100 PEs and with arrays A and B of size 2048 the instruction $C \leftarrow A + B$, took 31 milliseconds in parallel. The same instruction took 80 milliseconds on the VAX.

4.2 Reduction Simulation

The timings obtained for the reduction simulation are shown in Table 5. The results were plotted and the graph is shown in Figure 3. The following sequence of APL instructions was executed by the simulation program and by Purdue/EE APL.

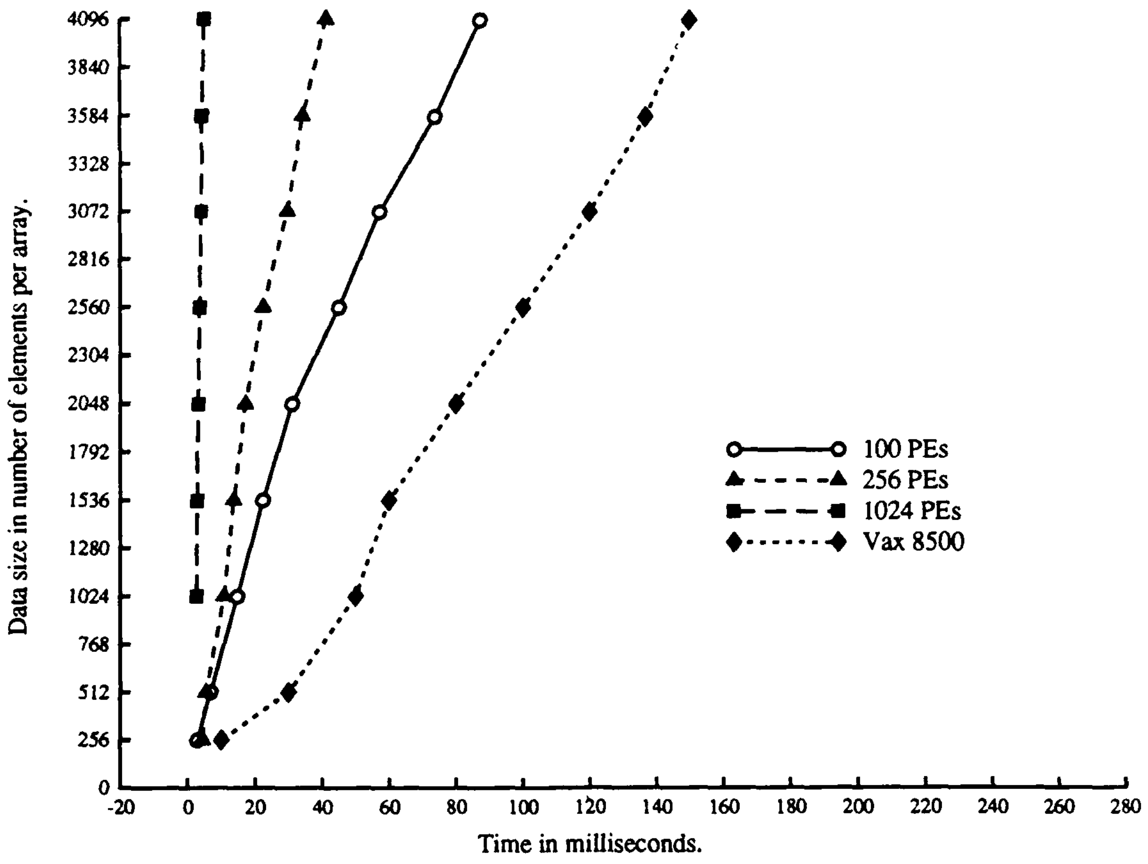

Figure 2: Time versus "data size" for the APL dyadic operation

Example 4.2 $A \leftarrow \iota x$
$B \leftarrow +/A$

Timings were obtained for the reduction instruction in a manner similar to that as described for the dyadic operation.

4.3 Subscript Simulation

The timings obtained for the subscript simulation are shown in Table 6. The results were plotted and the graph is shown in Figure 4. The following sequence of APL instructions was executed by the simulation program and by Purdue/EE APL.

Example 4.3 $A \leftarrow \iota x$
$B \leftarrow A$
$A \leftarrow \phi A$
$C \leftarrow B[A]$

The timings were obtained for the fourth step.

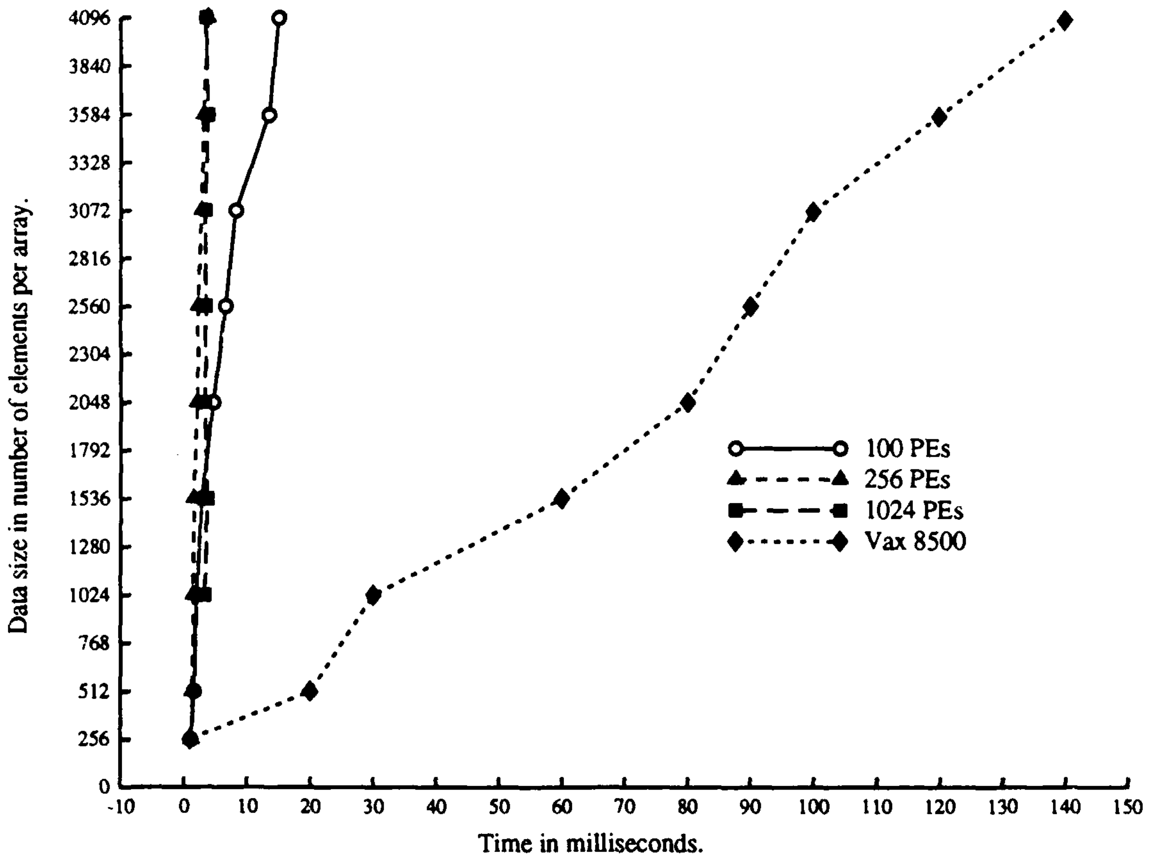

Figure 3: Time versus "data size" for the APL reduction operation

5 Analysis of Simulation Results

In this section the results presented in Section 4 are analyzed. First, we will focus on the dyadic operator. Table 4 shows the timings obtained for the dyadic operator and the plot for the data is shown in Figure 2. The first thing to notice about Figure 2 is that meshes of dimension 10, 16 and 32 did obtain speedups over the VAX. The speedups obtained for the mesh with 100 PEs for different data sizes were about a factor of 3 over the VAX; the speedups obtained for the mesh with 256 PEs were about a factor of 4 over the VAX; the speedups obtained for the mesh with 1024 PEs were about a factor of 30 over the VAX. Notice, that in going from 100 PEs to 1024 PEs the speedups achieved were increased by about a factor of 10. Although the observed speedups were not as great as expected, the speedups achieved for larger size meshes seemed to be about the order of the dimension of the mesh.

Table 5 shows the timings obtained for the reduction operator and the plot for the data is shown in Figure 3. As with the dyadic operator the first thing to note about Figure 3 is that speedups over the VAX were obtained. In fact, the speedups obtained for the reduction operator are slightly better than those obtained for the dyadic operator. For large enough meshes with large data arrays, the communication cost needed to perform the reduction operator would become great enough so that the reduction operation would require more time than the dyadic operator. The

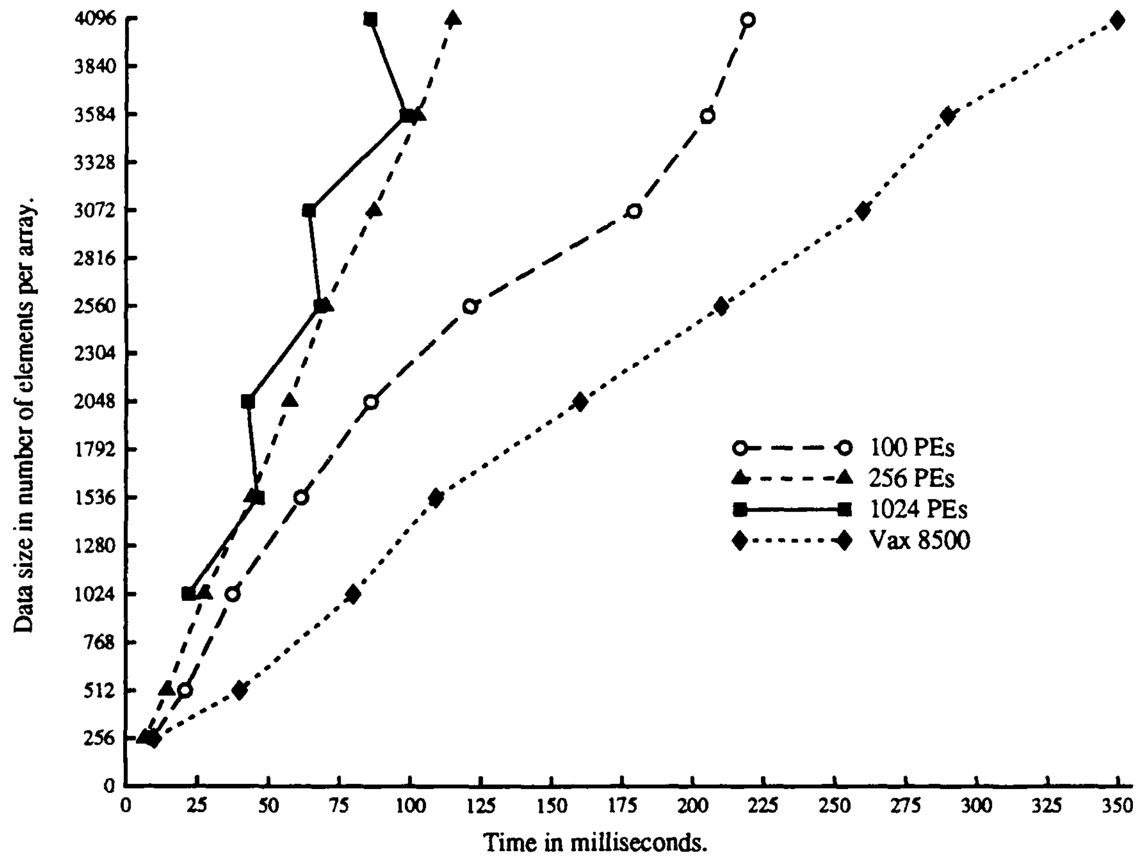

Figure 4: Time versus "data size" for the APL subscript operation

speedups obtained for the mesh with 100 PEs for different data sizes were about a factor of 15 over the VAX; the speedups obtained for the mesh with 256 PEs were about a factor of 40 over the VAX; the speedups obtained for the mesh with 1024 PEs were also about a factor of 40 over the VAX. The data sizes tested for the mesh with 1024 PEs were not really large enough to achieve the full benefit of the Reduction Algorithm. However, since the cases with 100 PEs and 256 PEs illustrate our point quite well we did not run the larger time consuming experiments for the case of 1024 PEs. In addition, we attribute the zig-zagging back and forth of the timings for the case of 1024 PEs to the symbol table searches. Unsuccessful searches required more time to execute than successful ones.

Table 6 shows the timings obtained for the subscript operator and the plot for the data is shown in Figure 4. Figure 4 shows that for the various values of mesh sizes we obtained speedups over the VAX in all cases. The speedups obtained for the mesh with 100 PEs were about a factor of 1.5 over the VAX for different data sizes; the speedups obtained for the mesh with 256 PEs were about a factor of 3 over the VAX; the speedups obtained for the mesh with 1024 PEs were about a factor of 4 over the VAX. Note, the speedup estimate for the case of 1024 PEs was based on the largest experiment. Since the large experiments required lots of computer resources, this limited the size of the simulations. However, notice in going from a mesh of size 100 to a mesh of size 256 our timings were sped up by about a factor of 2.5. Therefore, it seems reasonable to conclude that for large data sizes on the mesh with 1024 PEs that we could expect about a factor of 15 speedup over the VAX. Although a factor of 15 may not seem like much for a program that requires only milliseconds to run, consider a program that requires 15 hours to run on the VAX.

Since the APL reduction operator is monadic, the term "data size" has a different meaning than for the dyadic plus operator and also for the subscript operator. We can think of the subscript operator as having two arguments, the index vector and the subscript vector. We choose to represent the data size for the dyadic operator as the length of one of the arrays to be operated on and for the subscript operator we choose to use the size of the index vector.

For the experiments the times for the operators compared from least to most as follows: reduction, dyadic and subscript. Although the subscript operation was expected to take the greatest amount of time, we did not expect the reduction operation to be faster than the dyadic operation. However, there are two factors that probably account for this difference. As mentioned previously the dyadic operator requires no global communication among processors in the mesh, whereas, the reduction operator does. However, since we are basically charging the cost of an assignment statement for our communications, the timings for the reduction operator are somewhat optimistic. The other factor is that for the dyadic operator two values need to be looked up in the symbol table for every one looked up in the reduction operation. This is because of the definition of data size we adopted. Notice, also that in the sequential case the dyadic operation was slower than the reduction operation. For very large mesh sizes though, the analysis and the trend of the data indicates that the dyadic operation will eventually run faster than the reduction operation.

6 Conclusions and Further Research

The original goal in conducting this research was to determine whether or not speedups for a shared memory model sequential language could be obtained by implementing the language on a non-shared memory SIMD parallel computer. We focused on the language APL because of its inherent parallelism and on the 4-connected mesh computer because of its simplicity. We implemented on a simulated mesh the APL primitive scalar dyadic, reduction, and subscript operators, which studies [SW75] indicate comprise over 90% of "typical" APL programs. The timings obtained for the operations all indicated that the operations could be sped up using parallelism as the theoretical analysis conducted indicated.

Although only three APL operators were implemented, we implemented the three that comprise a large percentage of most APL programs. In addition, using known techniques [Abr70] [Bre82] in which one operator can be rewritten as some combination of other operators programs might be further simplified. The subscript operator being one of the most general operators can serve to rewrite many other operators. As further research, it would be useful to build a simulation program that implemented all APL operators so that complete APL programs could be run. Our research indicates that this would yield promising results.

The Amdahl's law analysis conducted in Section 2 indicated that one should not be overly optimistic about "parallelizing" a sequential language. The simulation conducted seems to indicate that we can speed up a majority of typical APL programs by more than a constant factor. Therefore, additional processors in the parallel machine would translate into significant additional speedups.

The Saal and Weis [SW75] study conducted on APL programs was done statically. A statistic that would be useful to obtain would be dynamic array size. If the dynamic operator usage percentages were known and also the array sizes being operated on, then it would be possible to estimate program performance in a parallel environment using our simulation results. We feel these results would be significant and interesting.

Other avenues to pursue for further research would be to try and implement APL on an actual mesh instead of doing a simulation on a sequential computer. Performing simulations on other connection networks such as the Shuffle Exchange [Sto70] or the Cube Connected Cycles [PV81] might also yield some interesting results. For the simulations we assumed the data was stored in row major order, it would be interesting to test out other data allocation schemes as well. In [Sch87] several data allocation schemes were tested for the n-cube.

References

[Abr70] P. Abrams. *An APL Machine.* Technical Report SLAC Report 114, Stanford University, 1970.

[AKPW83] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependencies to data dependencies. In *Proceedings 10th ACM Symposium on the Principles of Programming Languages*, pages 177–189, ACM, 1983.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, AFIPS, 1967.

[Bea68] G. Barnes and et al. The ILLIAC IV computer. *IEEE Trans. on Computers*, C(17):746–757, 1968.

[Bre82] N. Brenner. APL on a multiprocessor architecture. *APL Quote Quad*, 13(1):57–60, 1982.

[Bud84] T. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[CBHH87] J. Cuny, D. Bailey, J. Hagerman, and A. Hough. *Simple Simon: Testbed for Parallel Programming Environments.* Technical Report, University of Massachusetts, Amherst, Massachusetts, 1987.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *ACM Symposium on Theory of Computing*, IEEE Computer Society, 1978.

[Gol77] L.M. Goldschlager. *Synchronous Parallel Computation.* Technical Report TR-114/77, University of Toronto, 1977.

[GR76] L. Gilman and A. Rose. *APL an Interactive Approach.* Wiley, 1976.

[Ive62] K. E. Iverson. *A Programming Language.* John Wiley and Sons, New York, 1962.

[KKLW80] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorer for pipelined processors. In *4th International Computer Software and Application Conference*, pages 709–715, IEEE, 1980.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., New Jersey, 1978.

[Nel87] P. A. Nelson. *Parallel Programming Paradigms.* Technical Report, University of Washington, July 1987.

[PV81] F. Preparata and J. Vuillemin. The cube connected cycles: a versatile network for parallel computation. *CACM*, 24(5):300–309, May 1981.

[Sch87] J. Schaad. *Allocation Strategies for APL on the CHiP Computer.* Technical Report 87-03-06, University of Washington, March 1987.

[Sny84] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27–36, July 1984.

[Sto70] H. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. on Computers*, C-20(2):153–161, Febuary 1970.

[SW75] H. Saal and Z. Weis. Some properties of APL programs. *APL '75 Conference Proceedings (Pisa. Italy)*, :292–275, June 1975.

[Ull84] J. Ullman. *Computational Aspects of VLSI.* CSP, 1984.

[Upf84] E. Upfal. Efficient schemes for parallel communication. *JACM*, 31(3):507–517, 1984.