Staggered Distribution: A Loop Allocation Scheme for Dataflow Multiprocessor Systems

Joford T. Lim and A. R. Hurson
The Pennsylvania State University
Dept. of Elect. and Comp. Eng.
University Park, PA 16802

Ben Lee
Oregon State University
Dept. of Elect. and Comp. Eng.
Corvallis, OR 977331-3211

Behrooz Shirazi
The University of Texas at Arlington
Comp. Sci. Eng. Dept.
Arlington, TX 76019

Abstract

It has been shown that in many instances the run-time overhead of detection and allocation of dynamic parallelism in a program can easily offset the performance gain. Therefore, to improve performance and reduce run-time overhead, it would be logical to devise an allocation scheme which detects dynamic parallelism during compile-time. However, the difficult task of accurate estimation of the run-time parallelism is a stumbling block to this direction.

As a compromise, we propose an allocation policy which: i) detects dynamic parallelism for loop constructs during compile-time and, ii) allocates them to the estimated hardware resources in a staggered fashion using a set of heuristic rules. This paper introduces the proposed *Staggered Distribution Scheme* and addresses its simulation and performance improvement.

I. Introduction

Literature has addressed various concurrent programming techniques in order to partially satisfy the intensive computational demands of disciplines such as oceanography, astrophysics, seismology, meteorology, as well as atomic, nuclear, and plasma physics. Multiprocessing is one such technique which has been the subject of many recent researches.

Architects of such an organization must address the loss in processor efficiency due to two fundamental issues: memory latencies and synchronization overhead [3]. Traditionally, the detrimental effect of memory latency has been reduced by instruction pipelining. However, the restriction of a single thread organization of von Neumann-type architecture does not allow more than a few instructions in the pipeline. Moreover, the conventional techniques used to reduce the memory latency tend to increase the cost of task switching. Finally, the synchronization cost of the conventional multiprocessors makes the fine-grained decomposition of a program counterproductive.

The dataflow model of computation is different from the conventional control-flow model in that dataflow operations are asynchronous - i.e., the execution of an instruction is based on the availability of its operands. Therefore, instructions in the dataflow model do not impose any constraints on sequencing except the data dependencies in the program. Theoretically, in a dataflow machine, maximal concurrency can be exploited, constrained only by the availability of hardware resources. Dataflow machines treat each instruction as a task, and by applying a small synchronization cost, offer the ultimate flexibility in scheduling instructions. Dataflow machines may be regarded as an extreme example of machines with multiple threads, in which each instruction constitutes an independent thread, and only non-suspended threads are scheduled to be executed.

The dataflow model of computation has been a subject of study for over twenty years. Basic studies of dataflow computing have been promoted by various research groups and a number of hardware prototypes have been built and evaluated [4, 7, 11, 15]. However, before dataflow computers can become a viable alternative to conventional control-flow computers, several major drawbacks must be overcome [6]. This paper addresses the problem of detection and allocation of dynamic parallelism in a dataflow multiprocessor system.

The paper is organized as follows: Section II addresses the allocation problem and some of the schemes proposed in the literature. In Section III, the issue of handling dynamic parallelism is discussed, the DOACROSS construct is presented, and a modified DOACROSS construct for dataflow is shown. In Section IV, a scheme to allocate doacross loops called *Staggered* distribution is introduced. Section V shows the effectiveness of this scheme and analyzes the results of simulations. Finally, Section VI gives a brief summary and direction for future research.

II. Allocation problem

Similar to the control-flow multiprocessors, the issue of task allocation is also of major interest to dataflow multiprocessors. Two main approaches, namely *static* and *dynamic*, exist for task allocation [13]. In *static* allocation, the tasks are allocated at compile-time using

0-8186-2772-7/92 $3.00 © 1992 IEEE

global information about the program behavior and the system organization. The cost of allocating tasks is incurred once for a given program even though the program may be executed repeatedly. However, static allocation policies can be inefficient when estimates of run-time dependent characteristics are inaccurate. A *dynamic* allocation policy on the other hand is based on measuring processor loads at run-time, assigning activated tasks to the least loaded processor and balancing the load by migrating tasks. The disadvantage of dynamic allocation is the overhead involved in determining processor loads and allocation of tasks at run-time. The goal of program allocation is to maximize concurrency in a program graph by minimizing contention for processing resources. It has been shown that obtaining an optimal allocation of a graph with precedences is NP-complete [12]. Therefore, heuristic solutions are the only possible approach to solving the allocation problem suboptimally in polynomial time.

A number of heuristic algorithms have been developed for the allocation problem based on *list schedules* [1]. The most widely used schemes are the *critical path* heuristic algorithms. The basic idea is to assign each node of a directed graph a weight that equals the maximum execution time from that node to an exit node. An ordered list of nodes is constructed according to their weights, which is then used to dynamically assign nodes with highest weights to processors as they become idle. Ravi *et al.* [13] and Sarkar and Hennessy [14] have extended critical path heuristic algorithms by considering the effect of communication costs among processing elements (PE).

To determine the proper compromise between computation and communication costs, Lee *et al.* proposed the Vertically Layered (VL) allocation scheme [8]. The basic idea is to use heuristic rules to arrange the nodes of a dataflow graph into vertical layers, where each vertical layer represents a group of data dependent nodes, and then allocate each layer to a processor. Furthermore, it attempts to optimize the allocation by considering the inter-PE communication costs. This is done by considering whether the inter-PE communication overhead offsets the advantage gained by overlapping the execution of two subsets of nodes in separate processing elements. This process is repeated in an iterative manner until no improvement in performance can be obtained by combining vertical layers.

III. Dynamic parallelism

Performance of the aforementioned allocation policies can be further improved by considering the issue of handling dynamic parallelism [8]. Dynamic parallelism can be found in several sources: procedure invocation, loops and recursion. Dynamic dataflow architectures, which permit simultaneous firing of multiple instances of a node, could exploit dynamic parallelism by properly distributing the instances of these sources among processing elements.

Experience has shown that a significant overhead could be incurred in handling dynamic parallelism during run-time. However, this overhead can be reduced by using different techniques. For example, procedures can be allocated to processors during compile-time via inline expansion. Since loops are the largest source of dynamic parallelism, and recursion can be converted to loops, the rest of this paper is devoted to investigating dynamic parallelism within the scope of loops.

III.1. The DOACROSS model

Cytron [5] developed the DOACROSS model for the execution of loops with some degree of parallelism among various iterations. In this model, each iteration is assigned to a virtual processor, where execution of two successive virtual processors is delayed with d time period. In general, the delay d can range from zero (the vector loop case) to T (the sequential loop case), where T is the execution time of one iteration of the loop. The total execution time for a DOACROSS loop L of n iterations is:

$$TE(L) = (n - 1)\, d + T$$

This simple model, however, does not take into consideration the effect of inter-processor communication cost. As a result, during run-time some of the processors could be idle longer than the anticipated delays. To remedy this shortcoming, we introduce, in the following section, a modification of this model, which i) takes into account inter-processor communication cost and, ii) is targeted for a dataflow environment.

III.2. DOACROSS loops for dataflow

In our model, a dataflow graph is used to represent a loop construct, where the nodes represent the instructions and the arcs represent the data dependence among the nodes. This is shown in Figure 1, where F is the loop body.

If node i of one iteration (l) is data dependent on the result of node j of the preceding iteration (l - 1), where node i precedes node j, then there is a *lexically backward dependency* (LBD) [5] between iteration l and l - 1. This type of dependency contributes to a loop's delay. If we assume unlimited number of processors and if each iteration is assigned to a processor, then the total execution time of the loop in the presence of inter-PE communication cost will be:

$$TE(L) = (n - 1)\, t(i, j) + (n - 1)\, C + T = (n - 1)\,(t(i, j) + C) + T$$

where n is the number of iterations, $t(i, j)$ is the sum of the execution time of all the nodes from i to j that contribute to the final result of j, C is the inter-PE communication cost and T is the execution time of one iteration.

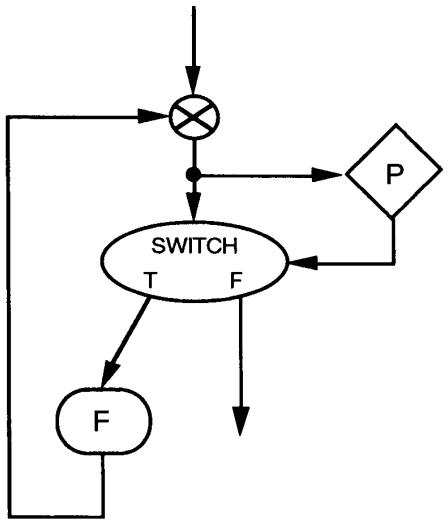


Figure 1. A loop schema.

If we have a limited number of processors P and the iterations are equally distributed among the P processors, then the delay due to the inter-PE communication will be $(P - 1) * C$. In the following section, we present a scheme to allocate/distribute DOACROSS loops among processing elements, taking the communication delays into account.

IV. Staggered distribution

The number of iterations of a loop can be either determined at compile-time or during run-time. In the first case, the number of iterations is fixed and known in advance. Hence, it can be unrolled according to the number of iterations. For the latter case, the number of iterations is not known, which means that unrolling during compile-time could result in inefficient resource utilization. This section will concentrate mainly on handling the first case.

IV.1. Unrolling effect

Unrolling the loop based on the number of iterations results in reducing the loop overhead, exposing more parallelism and decreasing the loop critical path. To show the effect of unrolling a loop, we generated parallelism profiles of some dataflow graphs using Id World [10].

The parallelism profile of a dataflow graph for a given input is a function ($pp(t)$), which determines the number of operators executed at each step t on an ideal machine [2]. The *ideal machine* has the following characteristics:

1. All operators take unit time.
2. Any number of operations can be performed in a step.
3. Communication is instantaneous.
4. Each operator executes as early as possible, that is, as soon as all its input data are available.

This model assumes unbounded processor, storage, and communication resources.

A program graph of an inner product and its parallelism profile is shown in Figure 2. This graph shows that from one iteration to the next iteration: i) instructions 4, 5 and 7 are independent, and thus can be executed concurrently, and ii) instructions 1, 3 and 6 are involved in generating the next instance of the loop. Therefore, they do not allow the parallelism of the loop to be exploited effectively.

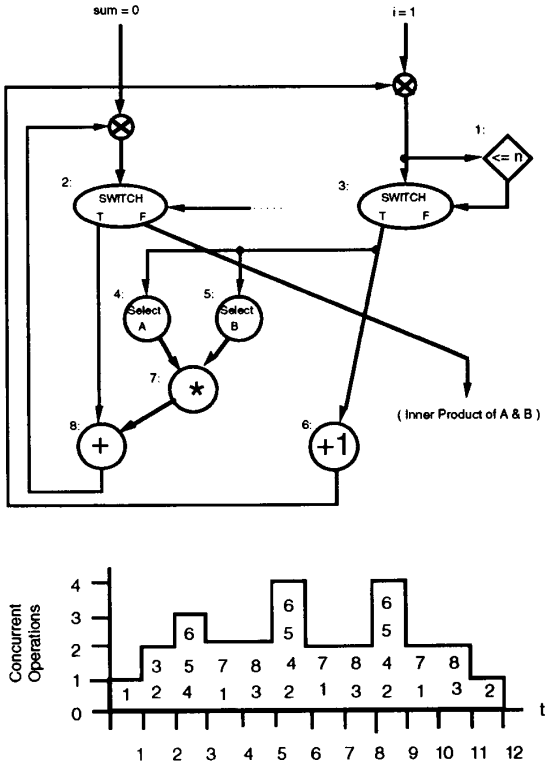


Figure 2. Program graph and parallelism profile for Inner Product, n = 3.

The step at which the computation completes is called the *critical-path length*, and is denoted by T_∞. Thus, T_∞ is the length of the longest chain of data dependencies in a program. The area under the curve $pp(t)$ gives the total number of operations executed, and is denoted by T_1. Using T_∞ and T_1, the *average parallelism* is defined as $AP = T_1/T_\infty$. For example, the parallelism profile of the inner product of length 3 (Figure 2) shows that the maximum parallelism obtained is 4, the critical-path

length is 12 and the total number of operations is 27, giving an average parallelism of 2.25.

An unrolled version of the program graph of Figure 2 is shown in Figure 3. Here as one can see all the iterations can start at the same time, the maximum parallelism is 6, the critical-path length is 6 and the total number of operations is 15, giving an average parallelism of 2.5.

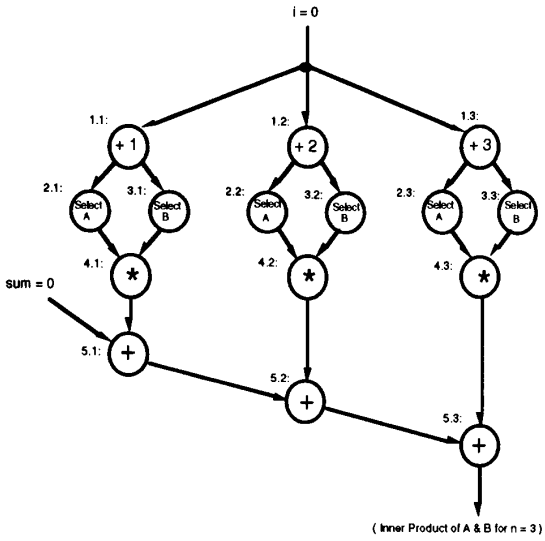


Figure 3. Unrolled version of Inner Product loop.

Although an *LBD* caused by instructions *5.i* ($1 \le i \le n$) is present, unrolling still increases the parallelism and decreases the loop's critical-path length. This effect is more evident as the number of iterations n is increased. Figures 4 and 5 show the parallelism profiles for the inner product and its unrolled version, respectively. Table 1 gives a comparison of the average parallelism for these two versions.

IV.2. Distribution/allocation of the unrolled graph

In this phase, we assume that the delay $t(i, j)$ is known or can be determined during compile-time. If k is the fraction of the delay $t(i, j)$ to the execution time of an iteration T, $k = t(i, j)/T$, then the fraction of T that can be executed concurrently in all iterations is $(1 - k)$. As mentioned in Section I, in a dataflow processor, each instruction constitutes an independent thread, and only non-suspended threads are scheduled to be executed. This means that a processor can switch between different iterations as long as some nodes are enabled. Therefore, if all iterations start at the same time and each processor is assigned the same number of iterations, then all processors would take the same amount of time to finish executing the $(1 - k)$ fraction of the loop. But each processor PE_i $(1 < i \le n)$, has to wait for processor PE_{i-1} to finish executing the k fraction and send the partial results to allow processor PE_i to continue execution. This creates delays due to the *LBD* and communication. Each processor is therefore idle for some time, which makes equal distribution an inefficient distribution scheme. It should be noted that for the sake of simplicity, we look at the nodes of an iteration as a single node - i.e., iteration node.

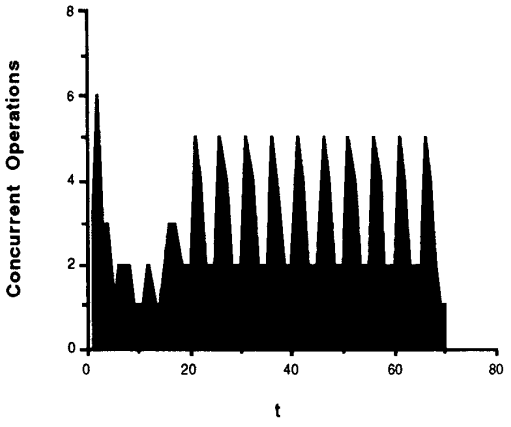
TABLE 1. Average parallelism of Inner Product & Unrolled Inner Product.

	T_1	T_∞	AP
Inner Product, n = 10	191	70	2.73
Inner Product, n = 30	491	170	2.89
Unrolled Inner Product, n = 10	71	18	3.94
Unrolled Inner Product, n = 30	191	38	5.03

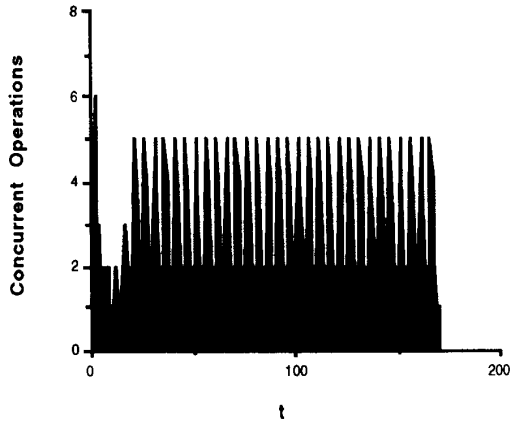
In our model, $k = 0$ implies no data dependence between iterations, hence no inter-processor communication. Therefore the iteration nodes would be distributed evenly among all PEs. If k is not equal to 0, the iteration nodes are distributed according to the following policy: The iterations assigned to PE_i succeed the iterations assigned to PE_{i-1} with PE_i having m more iteration nodes assigned to it than PE_{i-1}. This results in a distribution that increases as the iteration number/index increases. For example, if the number of processors is 3 and the number of iterations is 6, one possible distribution would be 1-2-3. This means that one iteration (the first) is assigned to the first PE, two iterations (the second and third) are assigned to the second PE and three iterations (the fourth, fifth and sixth) are assigned to the third PE. The delay caused by iterations assigned to PE_{i-1} will be equal to $k * T$ per iteration plus the communication cost C. This delay will be masked out by the $(1 - k)$ fraction of the additional iterations (m) assigned to PE_i. Incremental distribution of the iterations among processors is hence determined by:

$$m_i = \left\lceil \frac{n_{i-1} k T + C}{(1 - k) T} \right\rceil$$

where n_{i-1} is the number of iterations allocated to PE_{i-1}, T is the execution time of one iteration, k is the fraction of delay and C is the inter-processor communication cost.

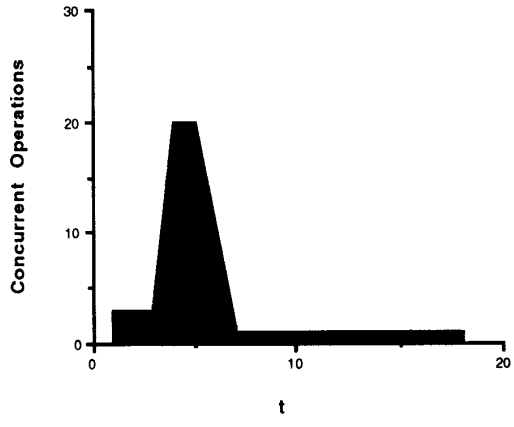


4a) n = 10

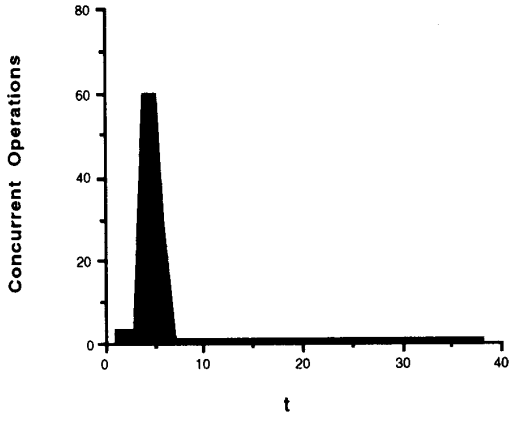


4b) n = 30

Figure 4. Parallelism profile of Inner Product.



5a) n = 10



5b) n = 30

Figure 5. Parallelism profile of unrolled Inner Product.

A DOACROSS loop ($k \neq 0$) implies an *LBD*. As a result, one should expect a delay of operations among the groups of iterations assigned to different processors and a communication delay since the partial result (tokens) has to be passed from one processor to another. *Staggered* distribution of iterations among processors will mask out these delays.

V. Simulation results

To evaluate the effectiveness of the *Staggered* distribution scheme, several loops with different execution times using the *Staggered* scheme and even distribution of iterations have been simulated and compared. One can expand the scope of the evaluation by considering other distribution policies [5]. Such a work is beyond the scope

of this presentation, but the interested reader is referred to [9].

Our simulation results are based on the following assumptions:

1. The underlying dataflow architecture is a distributed system.
2. The inter-PE communication delays are assumed to be constant, and the intra-PE delays are assumed to be negligible in comparison to the inter-PE communication delays.
3. The inter-PE communication delays are varied based on the ratio of *communication time to iteration execution time (C/T).*
4. Delays due to LBD are computed for various k values.

As mentioned before, the average parallelism (AP), is the ratio of the total execution time to the critical-path length. In our model, we have :

$$AP = \frac{n\ T}{(1 - k)\ T + n\ k\ T} = \frac{n}{(1 - k) + n\ k}$$

This equation shows that the average parallelism is independent of T.

Figure 6(a - d) depicts the number of PEs required to attain maximum speedup for various values of k. Both *Staggered* distribution (SD) and Equal distribution (EQ) for different *C/T* ratios are demonstrated. As can be seen, the proposed staggered approach offers better resource utilization and higher speedup than the traditional EQ approach. Both schemes require lesser number of PEs as the *C/T* ratio increases.

Figures 7a and 7b depict plots of the average parallelism (AP) and the maximum speedup (MS) for n = 300 and n = 1000, respectively. These plots show that the maximum speedup attained by using the SD scheme in the presence of inter-PE communication, is very close to the average parallelism, which can be considered as the maximum speedup possible for a particular loop. Moreover, an increase in the number of iterations n tends to give a better speedup.

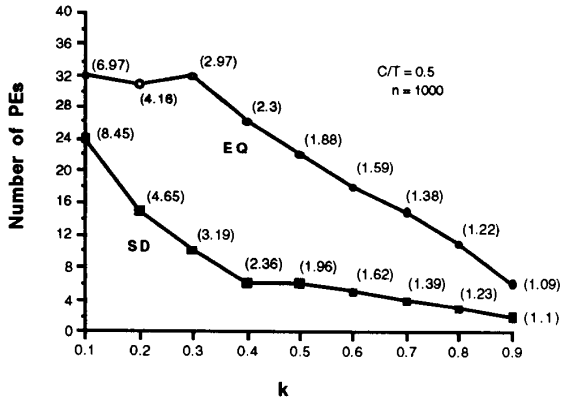
Table 2 shows the percentage improvement of the SD over EQ utilizing the same number of PEs as in Figure 6. The percentage improvement is generally higher for higher inter-PE communication cost and decreases as the amount of parallelism decreases (increasing k).

VI. Summary and future research

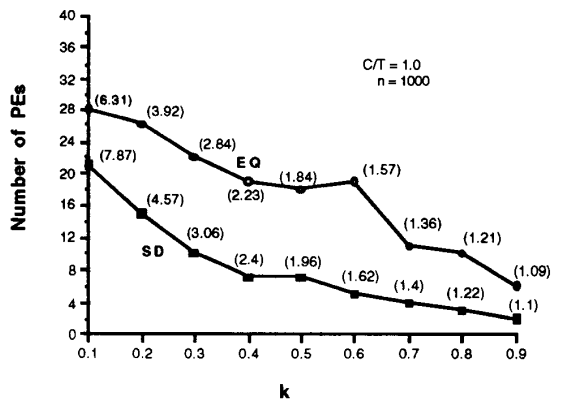
In this paper, we presented a distribution scheme for DOACROSS loops - i.e., *Staggered* distribution. The *Staggered* distribution scheme uses heuristics to distribute the loop iterations unevenly among processors in order to mask the delay caused by data dependencies as well as inter-PE communication. Simulation results have shown that our scheme is effective for loops that have a large degree of parallelism among iterations. Our scheme, due to its nature, distributes loop iterations among processing elements based on architectural characteristics of the underlying organization - i.e., processor speed, communication cost. In fact, the maximum speedup attained is very close to the maximum speedup possible for a particular loop even in the presence of inter-PE

TABLE 2. Percentage improvement of Staggered Distribution relative to Equal Distribution. Number of PEs in parentheses.

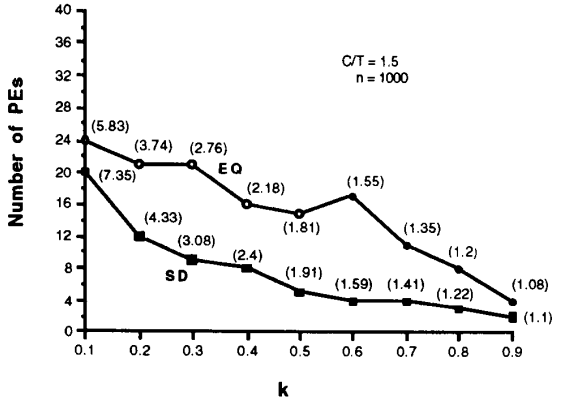
	n = 300				n = 1000			
k	C/T = 0.5	C/T = 1.0	C/T = 1.5	C/T = 2.0	C/T = 0.5	C/T = 1.0	C/T = 1.5	C/T = 2.0
0.1	20.6 (18)	22.46 (14)	24.31 (13)	24.83 (12)	20.28 (24)	21.75 (21)	21.56 (20)	22.37 (17)
0.2	20.02 (9)	18.28 (10)	18.97 (10)	20.7 (9)	17.24 (15)	17.92 (15)	18.35 (12)	19.56 (12)
0.3	16.99 (9)	17.01 (8)	15.9 (8)	17.81 (7)	16.42 (10)	13.72 (10)	16.6 (9)	15.21 (10)
0.4	12.96 (7)	12.7 (7)	15.56 (6)	16.25 (6)	15.55 (6)	15.11 (7)	14.11 (8)	14.31 (8)
0.5	11.55 (6)	12.78 (6)	14.58 (4)	11.17 (5)	12.89 (6)	11.79 (7)	13.7 (5)	13.98 (5)
0.6	10.19 (5)	9.39 (4)	10.96 (4)	11.57 (4)	9.38 (5)	9.65 (5)	11 (4)	11.33 (4)
0.7	9.96 (3)	8.26 (3)	7.59 (4)	9.01 (4)	7.15 (4)	8.23 (4)	8.79 (4)	9.69 (3)
0.8	6.9 (3)	4.58 (3)	6.46 (3)	6.99 (2)	6.29 (3)	5.48 (3)	6.05 (3)	5.49 (4)
0.9	4.03 (2)	3.5 (2)	4.33 (2)	3.48 (2)	4.05 (2)	4.31 (2)	4.15 (2)	3.89 (2)



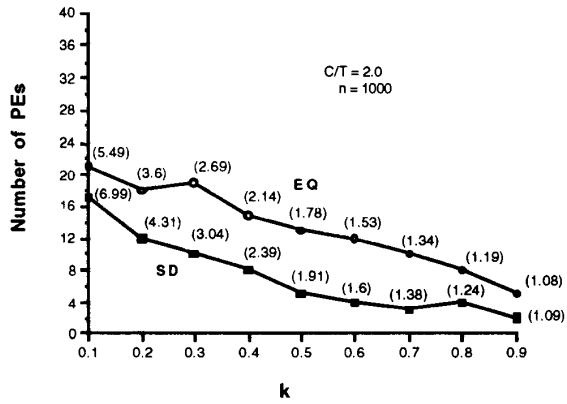
6a) n = 1000, C/T = 0.5.



6b) n = 1000, C/T = 1.0.



6c) n = 1000, C/T = 1.5.



6d) n = 1000, C/T = 2.0.

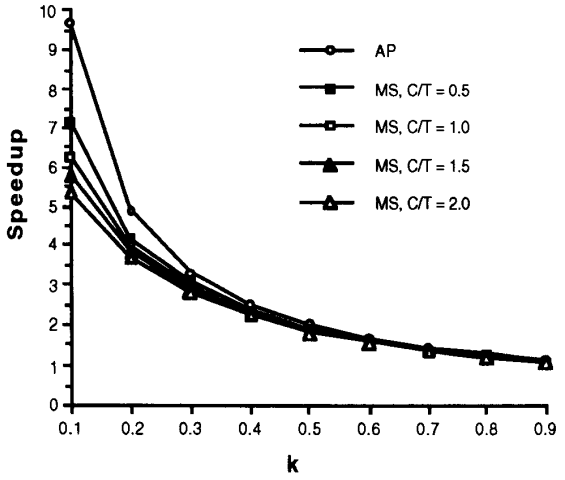
Figure 6. Number of PEs to attain maximum speedup.

communication cost. In addition, this scheme utilizes processors more efficiently, since relative to the equal distribution approach, it requires a lesser number of processors to attain maximum speedup. Although this scheme produces an unbalanced distribution among processors, this can be remedied by considering other loops when making the distribution to produce a balanced load among processors.

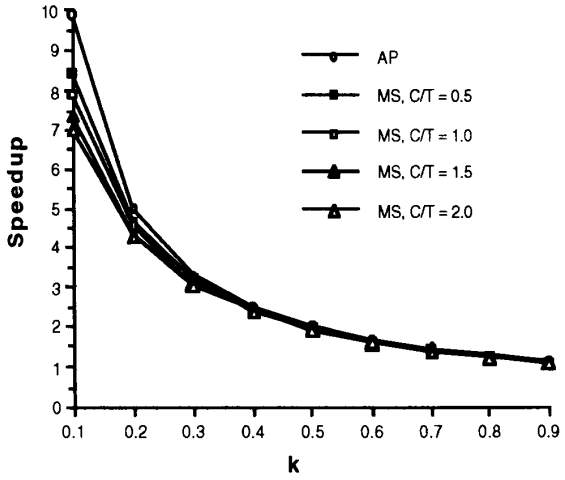
Naturally, for loops in which the number of iterations is not known in advance, unrolling is not possible during compile-time and hence the *Staggered* distribution cannot be used. But as soon as the number of iterations is known, the loop can be unrolled, and the *Staggered* distribution scheme can be used to distribute the loop iterations among the available processors. Obviously overhead in performing unrolling and distribution will be introduced. Further tests are therefore required to determine if the overhead introduced offsets the performance improvement attained by this scheme. As a next step, we intend to incorporate the proposed *Staggered* distribution scheme into the Vertically Layered allocation scheme [8].

VII. References

[1] Adam, T. L., Chandy, K. M. and Dickson, J. R., "A Comparison of List Schedules for Parallel Processing Systems." Communication of the ACM, Vol. 17, No. 12, Dec. 1974, pp. 685-690.



7a) n = 300



7b) n = 1000

Figure 7. Maximum speedup (MS) using Staggered Distribution.

[2] Arvind, Culler, D. E. and Maa, G. K., "Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs," The Int'l Journal of Supercomputer Applications, November 1988, pp. 10-36.

[3] Arvind and Ianucci, R. A., "Two Fundamental Issues in Multiprocessing," Proceedings 4th International DFVLR Seminar on Parallel Computing in Science and Engineering, June 1987, pp. 61-88.

[4] Arvind and Nikhil, R. S., "Executing a Program on the MIT Tagged-Token Data Flow Architecture," Proc. Parallel Architectures and Languages, Europe (PARLE), June 1987, pp. 1-29.

[5] Cytron, R., "DOACROSS: Beyond Vectorization for Multiprocessors," Proceedings International Conference on Parallel Processing, 1986, pp. 836-844.

[6] Gajski, D. D., Padua, D.A., Kuck, D. J. and Kuhn, R. H., "A Second Opinion on Data Flow Machines and Languages," Computer, Feb 1982, pp. 58-69.

[7] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Multiprocessor System," Journal of Parallel and Distributed Computing, Vol. 10, no. 4, Dec. 1990, pp. 309-318.

[8] Lee, B., Hurson, A. R., and Feng, T. Y., "A Vertically Layered Allocation Scheme for Dataflow Systems," Journal of Parallel and Distributed Computing, Vol. 11, 1991, pp. 175-187.

[9] Lim, J. T., and Hurson, A. R., "Staggered Distribution : A Loop Allocation Scheme for Dataflow Multiprocessor Systems," The Pennsylvania State University, Technical Report No. TR-92-098, June 1992.

[10] Nikhil, R. S., "Id World Reference Manual," MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA.

[11] Papadopoulos, G. M., and Culler, D. E., "The Explicit Token Store," Journal of Parallel and Distributed Computing, Vol. 10, No. 4, Dec. 1990, pp. 289-308.

[12] Polychronopoulus, C. D. and Banerjee, U., "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," IEEE Transactions on Computers, Vol. C-36, No. 4, April 1987, pp. 410-420.

[13] Ravi, T. M., Ercegovac, M.D., Lang, T., and Muntz, R. R., "Static Allocation for a Data Flow Multiprocessor System," 2nd International Conference on Supercomputing, May 1987.

[14] Sarkar, V. and Hennessy, J., "Compile-Time Partitioning and Scheduling of Parallel Programs," Proceedings SIGPLAN '86 Symposium on Compiler Construction, 1986, pp. 17-26.

[15] Yamaguchi, Y., Sakai, S., Hiraki, K., Kodama, Y. and Yuba, T., "An Architectural Design of a Highly Parallel Dataflow Machine," Proceedings IFIP Congress 1989, pp. 1155-1160.