Parallelism in Dataflow Programs[†]

Computation Structures Group Memo 279

December 15, 1987

Arvind
Gino K. Maa
David E. Culler

[†]Paper two in a sequence of four, including CSG Memos 278, 280, 281

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

Abstract

This paper develops a method for studying the inherent parallelism in "real" programs using a sequence of progressively more realistic dataflow execution models. The goal is to understand factors limiting speedup on parallel machines. It defines an ideal execution model and introduces *parallelism profiles* as a characterization of the inherent parallelism in programs. Although dataflow graphs provide a precise representation of parallel execution, our ability to generate graphs from a high-level language Id introduces a certain bias, which is examined. Profiles for several dataflow programs are presented, and one large program is examined in detail. Two approaches to deriving potential speedup are developed, one defines a more realistic finite-processor model which accounts for communication latency, the other predicts speedup through analysis of the parallelism profile. Finally, the effects of grouping portions of dataflow programs, such as function invocations or loop iterations, and requiring that the instructions in a group execute on a single processor are examined.

Parallelism in Dataflow Programs

1 Introduction

Scalability, the promise of faster execution as more processors and memories are added to the system, is an important aspect of parallel machines. However, when the observed speedup is less than optimal, one might conclude that the program does not have enough parallelism for machines beyond a certain size. What does this really mean? One possibility is that the algorithm does not offer sufficient parallelism. The algorithm may have parallelism, but it may be obscured by the language or the coding style. Perhaps a more sophisticated compiler could have found the parallelism. The program may have parallelism, but it may be of a form which the machine cannot exploit. Finally, the program may have parallelism of a useful form, but poor resource management or contention for physical resources may compromise it. A better understanding of how these various factors contribute to a lack of speedup is essential to developing large-scale parallel machines for general purpose computing. This paper illustrates a method for studying this question developed in the context of dataflow machines and presents preliminary data on the parallelism present in dataflow programs employing traditional algorithms.

The first step is characterizing the *inherent* parallelism in the program. This notion proves to be rather subtle, and in explaining it we must also explain what it is not. Formal techniques have been developed to show that exploiting parallelism in certain problems is difficult regardless of what algorithm is used. Notably, the AT^2 results found in VLSI theory demonstrate that if a problem is solved in parallel then a large amount of communication is required. While this work has clear merit, we do not yet see how to apply it to realistic applications. Highly parallel algorithms have been developed recently under a variety of parallel models of computation, such as PRAM, but in achieving such short critical paths the total amount of work tends to be large. As we are working with constant factor speedups, the total amount of work is critical. Certain architectures achieve tremendous parallelism, but only when algorithms are developed specifically for the particular architecture[13]. We do not assume that such novel algorithms be used, as many conventional algorithms have tremendous parallelism, but we require that the algorithm be expressed precisely. Real-world applications generally involve a number of algorithmic phases. Interactions between phases, end conditions, error analysis, and synchronization primitives often determine the parallelism in the program as a whole, so studying applications in full detail is essential. This raises certain difficulties because applications must be coded in some language and executed in some manner. For complete programs specified in full detail, we want to capture precisely what operations can be performed in parallel without introducing any constraints beyond the essential dependencies in the program.

Dataflow program graphs provide a precise description of parallel computation[9]. Arcs indicate data dependence between operations, and nodes not connected by arcs in the graph represent opportunities for parallel execution. We define an ideal execution model for dataflow graphs which executes all operators in parallel except as constrained by data dependencies. Based on this model, programs can be characterized by their *parallelism profile*, from which critical-path, average parallelism, and potential speedup can be deduced.[1]

[1]In a companion paper [10] we have presented a detailed account of the fraction of operators that represent the overhead or the cost of parallel execution. These two papers together provide a much more useful characterization of "useful" parallelism in dataflow programs.

Next, we try to characterize how this potential parallelism is compromised by various aspects of the implementation. The graphs considered are compiler generated from the language Id, so we examine the bias this introduces and restrict our focus to problems which are comfortably expressed in the language. By considering the behavior of the program under the constraint that only a certain number of operations can be performed simultaneously, we focus on potential speed-up, rather than potential parallelism, and show that the parallelism profile is an excellent tool for estimating potential speedup. We then consider the impact of communication latency, showing that as the latency increases more parallelism is required to achieve a given speed-up. Finally, we consider the effects of allocating fixed amounts of computing resources to portions of the computation.

Section 2 presents a graphical representation of programs, our idealized execution model, and parallelism profiles. Section 3 examines capabilities and limitations in generating graphs from a high-level language and explains certain subtleties in the model. Section 4 considers how much parallelism can be exploited effectively on a finite number of processors incurring memory and communication latency. Finally, Section 5 introduces program partitioning at various levels of granularity into the execution model and examines its effects on potential speedup. Throughout this paper we assume the reader has some familiarity with dataflow; a good introduction to the MIT Tagged-Token dataflow may be found in [6].

2 Parallelism Profiles under Ideal Execution

Dataflow graphs have a well-defined meaning without any timing assumptions, and we generally view dataflow execution as asynchronous. Nonetheless, in order to characterize the parallelism in dataflow programs we consider a synchronous unit execution model with unit time per operation. We are not, however, exploiting aspects of synchronous execution.

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

- All operators take unit time,
- Unbounded number of operations per step,
- Zero communication delay,
- Unbounded memory and communication resources,
- Eager scheduling of enabled operators.

We illustrate the method for generating parallelism profiles through an example. Figure 1 shows a program graph which computes the inner products of two vectors, A and B, of size n. Initially a token corresponding to sum with value zero is input to the left switch and a token corresponding to i with value one is presented to the right switch and the $\leq$ predicate. The value n as well as descriptors for the two vectors are *loop invariants* [10] and thus can be considered to be embedded in the graph. Assume that the value of n is 3. In step 1, instruction 1 (*i.e.*,the operator $\leq$,) fires producing tokens with value TRUE for the control inputs of the two switches. Instructions 2 and 3 fire in the second step and produce the tokens carrying the value of sum for instruction 8 and i for instructions 4, 5, and 6. In step 3, instructions 4, 5, and 6 execute while the token for sum waits for the other input to instruction 8. Firing of instruction 6 provides input to the predicate and the right switch. In step 4, the value of sum that has been waiting is added to the result of instruction 7 while the new value of i passes through the switch. Note in step 5 of the parallelism profile in Figure 1 that the second iteration has begun while the first is still active. Execution continues in this manner until step 14 when it produces a token on the False side of the Switch in instruction 2. The pattern in steps 4 to 6 covers all 8 instructions and repeats for every iteration. Note, a node is fired at the step corresponding to the maximum of the times of its input tokens.

The step beyond which $pp(t)$ is uniformly zero is called the *critical path length*, that is, the length of the longest chain of data-dependencies in the program. The area under the curve $pp(t)$ gives the total number of operations executed. The ratio of these is the average parallelism. It can be seen from the parallelism profile of the inner-product n = 3 shown in Figure 1 that the critical path length is 12 and the total number of operations is 27. Note, this describes the parallelism for this particular method of computing the inner-product and does not imply that this is the maximum achievable parallelism. However, other methods with more parallelism would be described by different graphs.

3 Generating Dataflow Graphs

Although dataflow graphs provide a precise representation of parallel execution, the utility of the model is limited by our ability to generate the graphs themselves. We will consider only graphs generated by a reasonably sophisticated compiler for the language Id. Id is a functional language extended with I-structures to provide efficient array manipulation. A broad class of algorithms are easily expressed in Id, although we recognize that it is difficult, perhaps impossible, to express certain types of algorithms in Id efficiently. This section explores the implications of the language and the compiler on parallelism profiles.

3.1 Compiler generated graphs

Inner-product may be expressed in Id as follows.

```
Def ip A B = {(l,h) = bounds(A) ;
              s = 0
              In
                {For j From 1 To h Do
                   Next s = s + A[j] * B[j]
                 Finally s }} ;
```

The dataflow graphs produced by the Id compiler are based on a fixed set of schema and rules for composition [19] which ensure deterministic behavior under all execution orders. Arithmetic expressions and let-blocks are described by acyclic graphs. Conditional expressions are constructed using switch operators to steer values to the appropriate arm, based on a predicate, allowing portions of the conditional to execute before all the inputs of the expression are present. Iteration is captured by a loop schema, which permits arbitrary overlap of iterations, unless constrained by data dependencies. Values which are arguments to an invocation of a loop but are constant over all iterations are not circulated, but are explicitly stored in a constant area (see [6] or [2]) in the

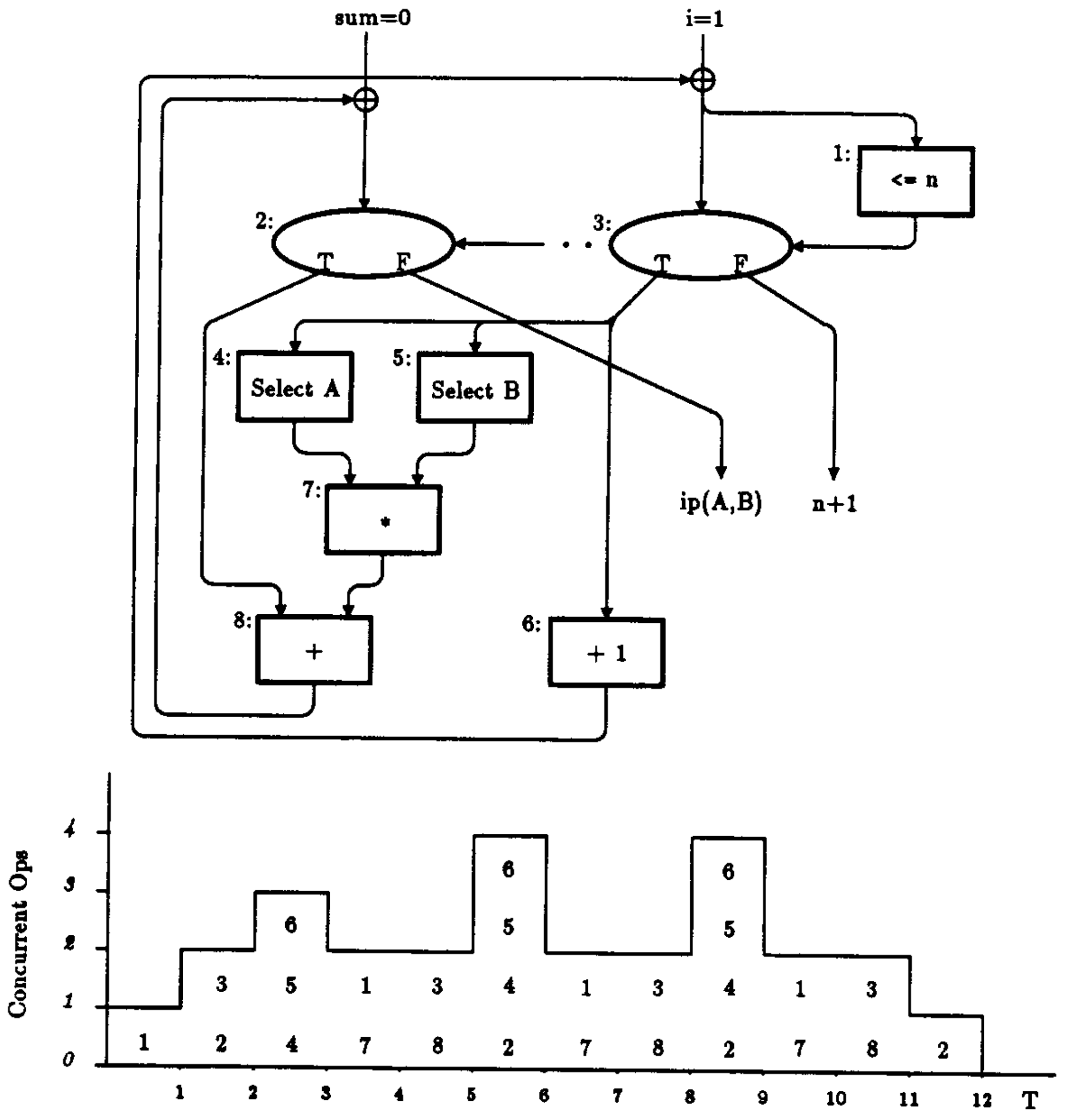

Figure 1: Program Graph and Parallelism Profile for Inner Product, n=3

loop pre-amble. User defined functions and loops are compiled into code-blocks, which are invoked by an application schema, permitting arbitrary recursion. The class of graphs generated in this manner are deterministic and self-cleaning.[5] Furthermore, graphs are embellished so each code block receives a trigger to enable nodes with constant input and produces a signal indicating that all nodes with no outputs have fired [19].

The graph generated by the Id compiler for ip, the inner product program, contains 31 instructions. Most are for setting up and cleaning up the loop and are thus executed only once. The graph for ip shown in Figure 2 though stylized captures the essential features of the compilation for drawing the parallelism profile. The output of the Id compiler can be executed on GITA [16], the Graph Interpreter for the Tagged-Token Architecture, which generates parallelism profile graphs as a part of its runtime statistics reports. As can be seen in the profile in Figure 2, compiler generated graph executes 5 more instructions per iteration than the graph in Figure 1. These additional instructions are generated for tag manipulation and control of loop unfolding, which is discussed further in a companion paper.[8] Graphs generated by the current Id compiler incur roughly 150% overhead in terms of the number of instructions executed beyond the essential computation in order to allow maximal parallelism[10] while preserving determinacy. We should note that although the ideal machine we consider here is synchronous, dataflow graphs specifically do not rely on any assumptions about the time required to perform operations or transmit tokens.

3.2 Associativity and Commutativity

The language does not assume associativity, commutativity, *etc.* of arithmetic operators, and therefore the compiler can not make use of these properties in generating graphs. The inner-product example forms a linear sum of product, so the graph is a simple loop. Forming a binary summation tree would offer more parallelism, and if this is the programmer's intent, a recursive formulation should be used.

Note that in the inner product example, the multiplications could complete in any arbitrary order, and the sum would still be formed in sequence. It is possible to describe graphs which would combine arbitrary partial products, in whatever order they happened to be produced, but again this is outside the language. Id intentionally has no means for expressing non-determinism; assuring determinacy under parallel execution is critical in developing correct applications. Still there are important application areas where non-determinism is essential and extensions to the language are being considered.

3.3 Non-strict Data structures

The language does not compromise parallelism in manipulating data structures. I-structures in Id are a form of non-strict array which is explicitly allocated and filled, with synchronization on an element by element basis. An attempt to read an element which is not present is deferred until the element is written and processed immediately thereafter. To ensure determinacy, an element can only be written once. By way of contrast, strict data structures[9] can be treated as a single value, but operations which access particular elements of the structure must wait until the entire array is defined. This prevents asynchronous production and consumption of structures much as does barrier synchronization. Approaches which detect parallelism in sequential programs are severely restricted by the difficulty of compile time analysis of subscript expressions; with I-structures no such analysis is required, as dependencies are resolved dynamically. With the help of

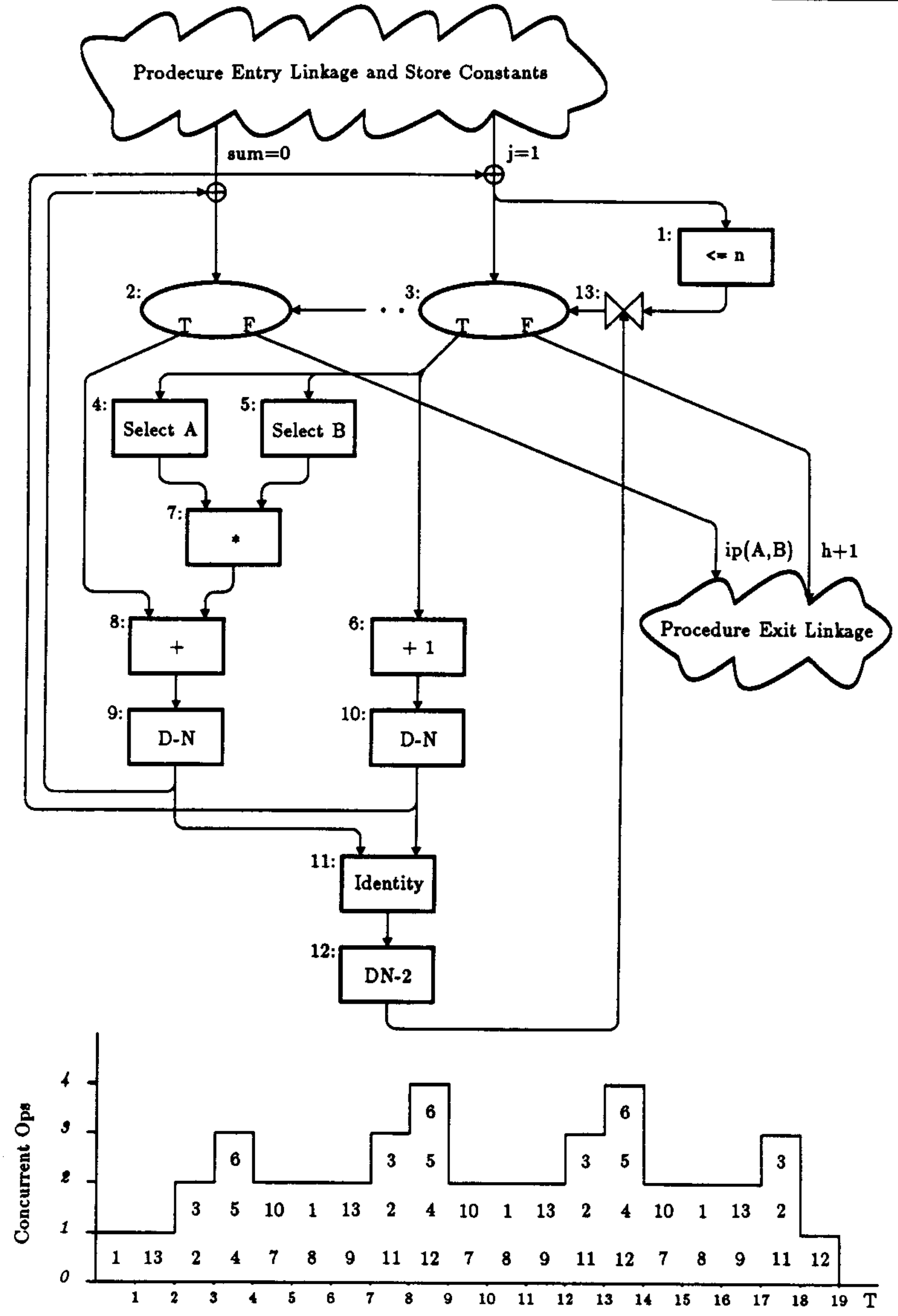

Figure 2: Compiler-Output Program Graph for Inner Product

following example, we show concurrent production and consumption of an array which is produced by summing two vectors and then "consumed" by the inner product program.

```
Def ip_vsum A B = ip (vsum A A) (vsum B B) ;

Def vsum A B = {C = array (1,n) ;
                 {For j From 1 To n Do
                    C[j] = A[j] + B[j]}
               In
                 C } ;
```

The vsum program implies no restrictions on the order in which the elements of array C are filled. Furthermore, the iterations of the loop are independent and can proceed in parallel. In most conventional languages, the best one could hope for is that the critical path length for ip_vsum would be the sum of the critical path lengths of ip and vsum. In Id, however, vsum can return the descriptor of C, its result vector, *as soon as it is allocated, even before its loop has completed filling it.* ip gets these descriptors, and can begin work immediately. Because arrays in Id have "I-structure" semantics, vsum and ip are automatically *pipelined*, working in tandem as producer and consumer respectively.[2]

Figure 3 shows the parallelism profiles for vsum, ip and ip_vsum respectively ($n = 10$). The critical paths and total operations for vsum are 63 and 158 respectively; for ip, they are 59 and 153, respectively. In ip_vsum, the individual profiles of vsum and ip are superimposed, rather than strung out in sequence. The critical path, instead of being $63 + 59$, is only 76, even though the instruction count is 500. The last number is greater than 469 ($= 153 + 2 \times 158$) because of the three procedure calls. Ideal profiles are indicated as in Figure 3 and having unbounded operations per step, $p = \infty$, and zero latency, $l = 0$ discussed below.

3.4 Parallelism in Nested Loops

Composition of control structures does not inhibit parallelism. Some approaches allow only outer loop parallelism or only inner loop parallelism, or similar restrictions. Id allows loops to unfold, functions to be spawned in parallel, etc., constrained only be data dependencies, irrespective of nesting of control structures.

Figure 4 gives the parallelism profile for matrix multiplication of two 16×16 matrices using the traditional algorithm, showing unfolding of nested loops. The upper profile counts *all* operations, and the lower profile counts only the floating point operations. The outer loop spawns 16 concurrent instances of the middle loop, each of which spawns 16 instances of the inner product. Unfolding of the inner product depends on the availability of elements of the input matrices. The critical path length is 291 and the total number of operations is 72,344, out of which 8192 are floating point operations. The average parallelism is 249 operations per step. The bell-shape of the profile arises because the way loops unfold causes inner loops to by staggered slightly, and the summation in the innermost loop is done sequentially. The rapid oscillations reflect the oscillating parallelism in the inner product. We could spawn the inner-loops in a tree, and perform the summations as a binary

[2]I-structure operations introduce a subtlety in the ideal model. The node which receives the result of an I-fetch operation does so one unit after the maximum of the time of the I-fetch and the time of the corresponding I-store.

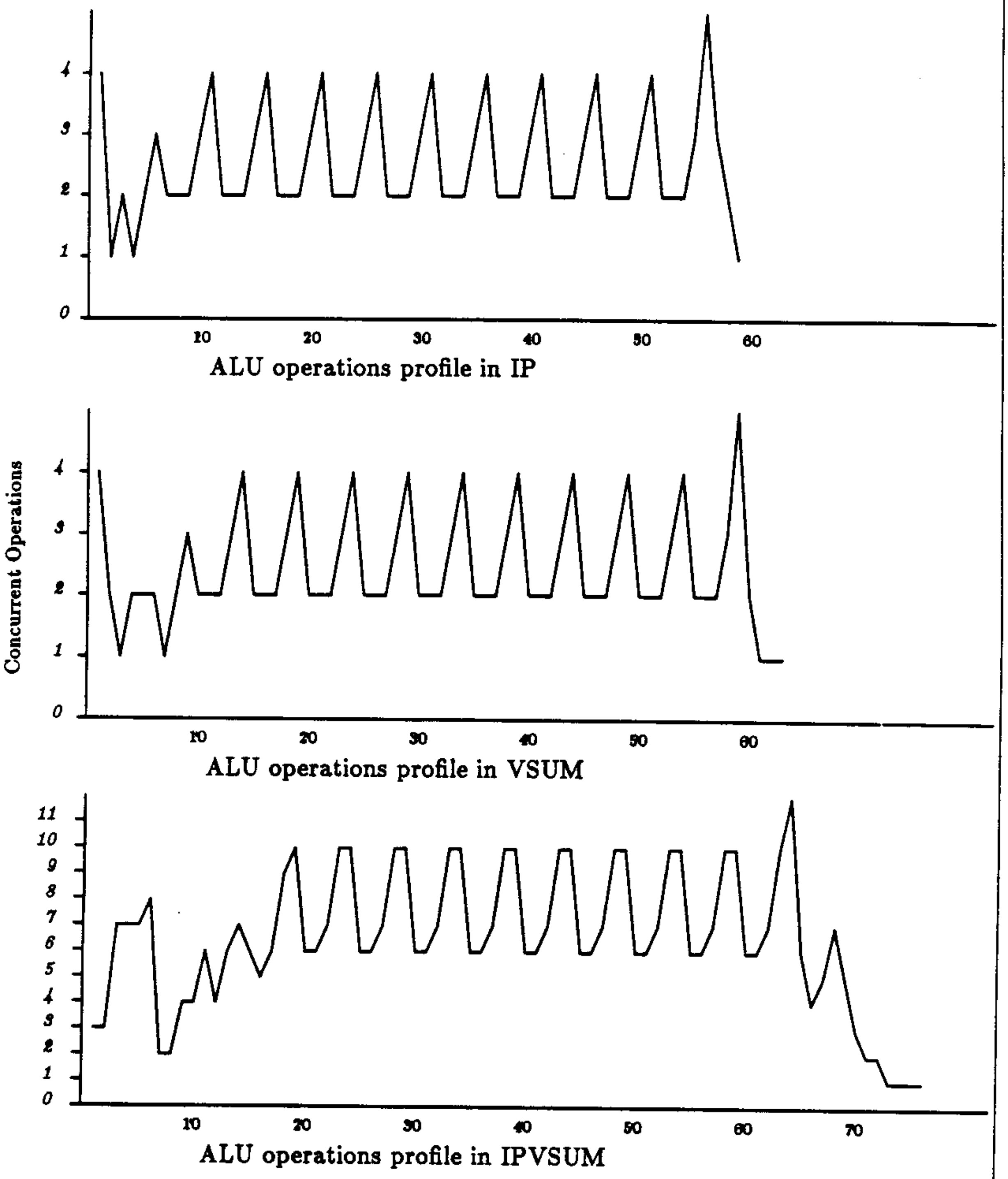

Figure 3: ($p = \infty, l = 0$) Parallelism Profiles for Vector Sum, Inner Product and their composition (size 10).

tree to reduce the length of the critical path further, but it is evident that substantial parallelism is present even in the traditional algorithm.

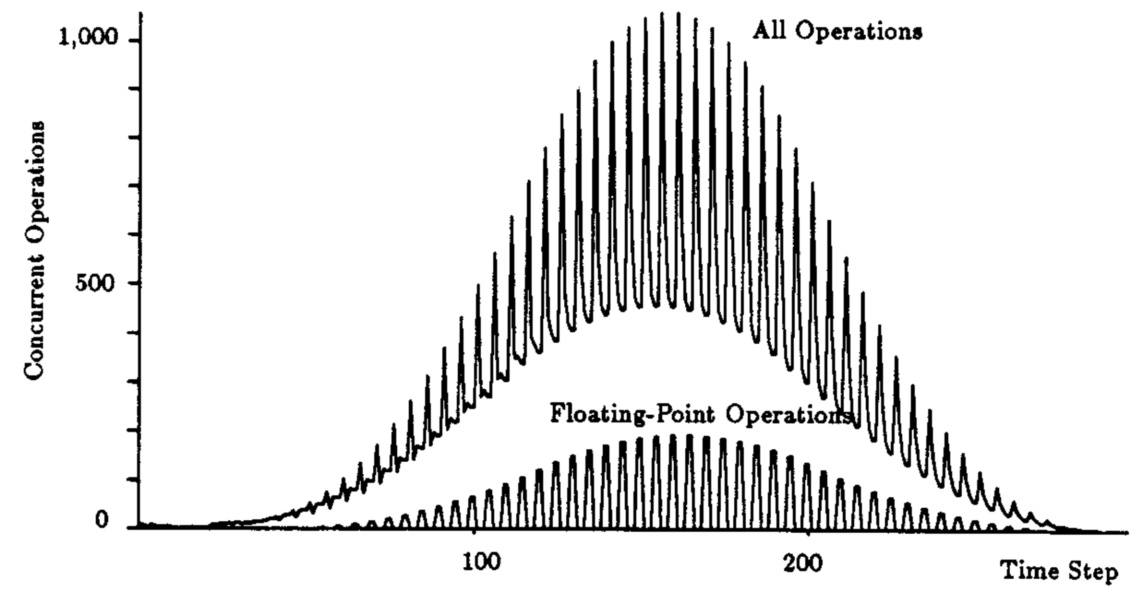

Figure 4: ($p = \infty, l = 0$) Parallelism Profile for Matrix Multiply (16 × 16)

Figure 5 shows the parallelism profile for a 16 × 16 LU decomposition, an important step in solving simultaneous linear equations (critical path = 1312 , total operations = 35,552). This example involves nesting of loops conditionals and user-defined functions. The descending stair-step behavior coincides with our intuition that after each pivot the remaining portion of the matrix has one fewer row and one fewer column.

These profiles accurately capture the inherent parallelism in programs, given the specific algorithms employed and the compilation technology. Factors which place additional constraints in any realistic implementation, such as as multiple simultaneous reads to an array component, communication latency, locality, contention, and distribution of work are abstracted away entirely and do not affect the parallelism profile.

3.5 Large Id Application Kernels: PIC and SIMPLE

In keeping with the goal outlined in Section 1 we now show that parallelism profiles can be generated and analyzed for complete programs. The two programs we consider are large with complex data and control structures, exhibiting non-trivial producer/consumer relationships.

Figure 6 shows the parallelism profile for an entire electrodynamics application (PIC) using a particle-in-cell approach. The idea is to model the movement of charged particles under the field they induce. This example shows 4 time steps with 640 particles in 64 cells (8 × 8 mesh); realistic problems use ≈ 1 million particles and 100,000 cells. For the problem in Figure 6, the number of operations executed is 1,573,733 and the critical path length is 3,399.

The parallelism profile shows that a large amount of parallelism is generated while computing the potential and solving Laplace's equation. This particular program uses a multigrid method to

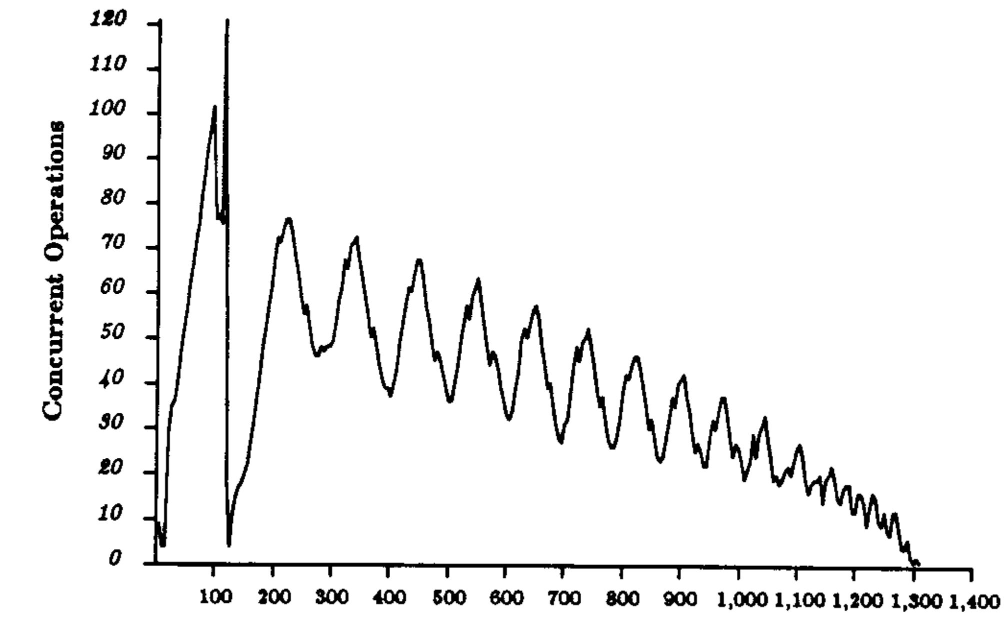

Figure 5: ($p = \infty, l = 0$) Parallelism Profile for LU Decomposition (16 × 16)

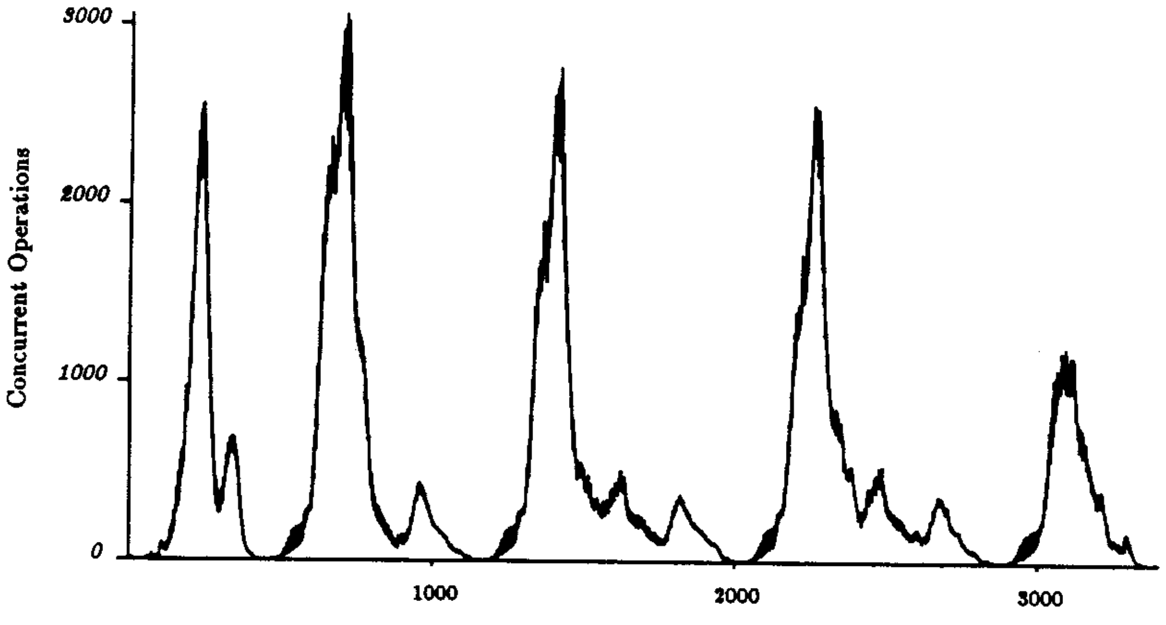

Figure 6: ($p = \infty, l = 0$) Parallelism Profile for Particle-In-Cell (4 iterations, 8 × 8)

solve Laplace's equation, which generates the first peak in the profile. Partially overlapped with this, but generating a second peak, is the calculation of the field and the resultant acceleration of each particle. Then there is a strong constriction point as the maximum acceleration and velocity are extracted and Newton's approximation is used to compute the time step. Once the time step is determined, the computation of new particle positions begins, generating substantial parallel activity. But note, the computation of the new charge density overlaps with the particle push, until a new time step is determined. After the second iteration, the behavior is periodic.

Figure 7 shows the parallelism profile of the SIMPLE code, a hydrodynamics and heat flow code that has been studied extensively both analytically [7] and by experimentation. A detailed discussion of the program appears in [4] and [10]. [3] This profile shows 3 iterations of the outer loop of SIMPLE on a 20 $\times$ 20 mesh, while a typical simulation run performs 100,000 iterations on 100 $\times$ 100 mesh. The critical path is 1976 and the instruction count is 1,471,374. As can be seen from iterations 2 and 3, there is no significant parallelism *between* the outer loop iterations of SIMPLE; the profile for N iterations can be obtained by repetition of a single iterations profile. To show how the profile changes with the size of the problem, we have drawn the profile for the 32 $\times$ 32 mesh in Figure 8. (Critical path = 1082, instruction count = 1,446,478.)

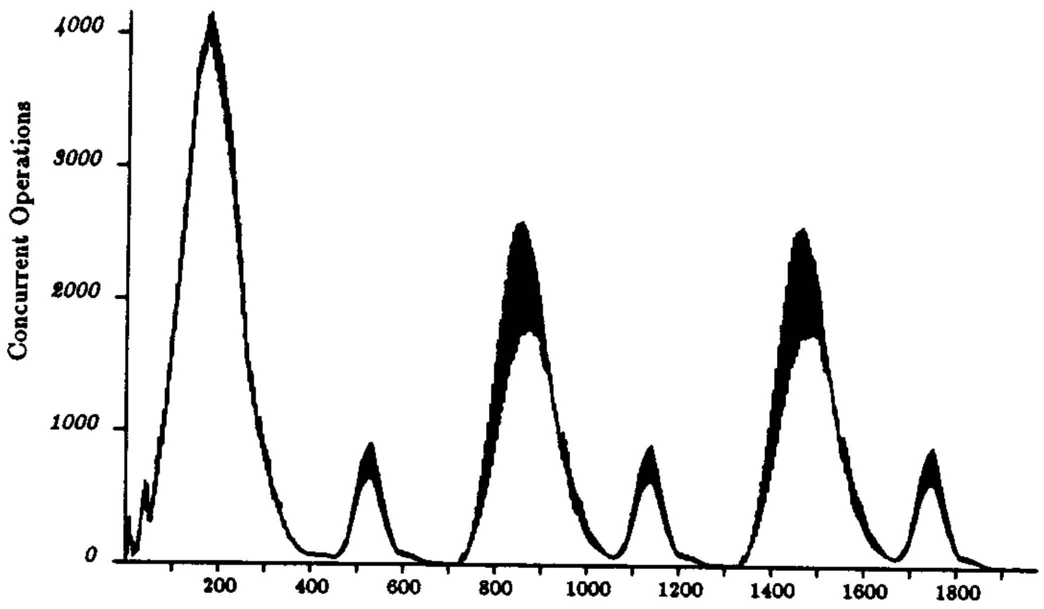

Figure 7: $(p = \infty, l = 0)$ Parallelism Profile for SIMPLE (3 iterations, 20 $\times$ 20).

It is noteworthy that the potential parallelism varies tremendously during execution, a behavior which in our experience is typical of even the most highly parallel programs. We believe that any large program that runs for a long time must have sufficient parallelism to keep hundreds of processors utilized; several applications that we have studied support this belief.

[3]The version of Simple used here is slightly different than that used in the bulk of [10]. It uses lower level abstractions and has minor algorithmic differences.

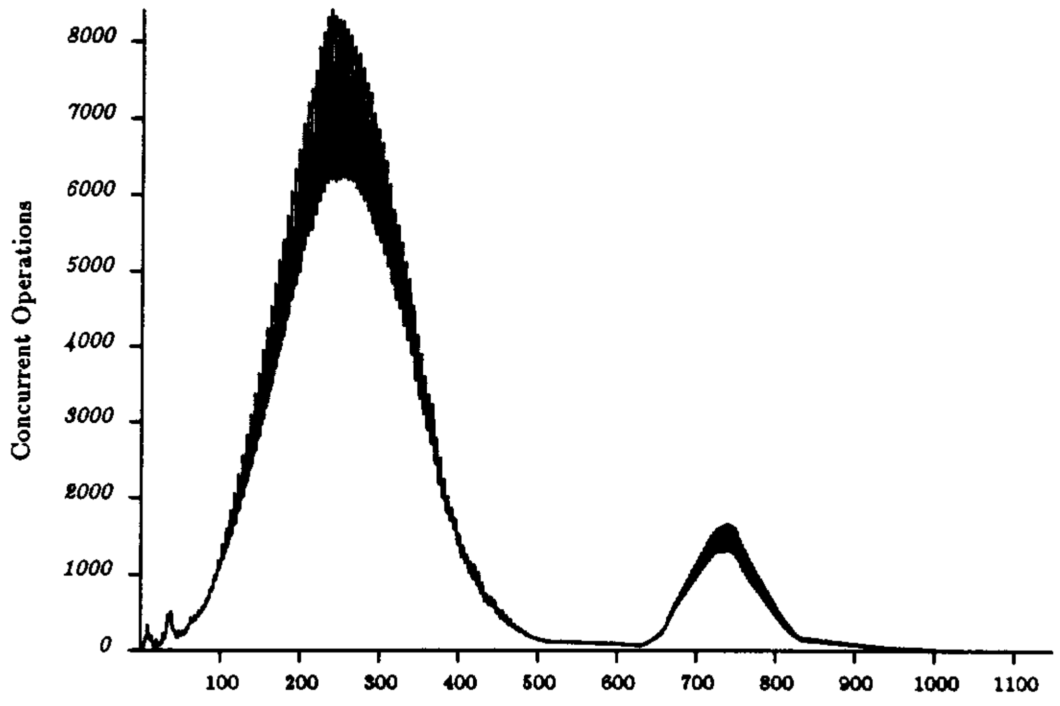

Figure 8: $(p = \infty, l = 0)$ Parallelism Profile for SIMPLE (1 iteration, 32 × 32).

4 Parallelism Profiles on Finite Number of Processors

Characterizing the inherent parallelism in programs is important insofar as it allows us to infer the behavior of the program on a finite number of processors. We are unlikely to be interested in the maximum number of processors potentially active at one time or to see execution times close to the critical path. Even average parallelism can be a misleading indicator of performance. However, as we consider the program behavior on a finite number of processors, we must discard our unrealistic assumption of zero communication latency, as it is impossible to build a machine that can always communicate an operand from the output of one node of a dataflow graph to the input of the destination node instantaneously. One reason is the internal pipelining of a processor and another is the delay in processor-processor and processor-memory communications. In most realistic designs, n processors and memories are interconnected by a multi-stage network, where the average latency of memory access is of the order of $log(n)$. Consider the best-case assumption that the communication network and processors are completely pipelined, *i.e.*,a processor/memory can send a message at every time step. To keep the pipelines full, processors need many instructions eligible for execution at each time step. If a processor is to avoid idling then the program must have sufficient parallelism to absorb memory access and communication latencies. In this section we present two approaches to characterizing the behavior of programs on a finite number of processors. The first modifies the ideal machine to constrain the program to execute a finite number of operations in each step, and the second estimates potential speedup from the parallelism profile itself.

4.1 The "Finite-Processor" Execution Model

The first step towards a realistic execution model is to bound the amount of available processing resources. Instead of allowing an arbitrary number of instructions to be executed during each step, a maximum limit of n operations during each step is imposed. This represents n processors only in a very abstract sense. We do not assign activities to processors, but rather choose up to n enabled activities in each step. One can view it as representing a dataflow system where the mapping of activities onto processors is perfect in that there can never be a situation where n operations are enabled, but not all n can be processed because some are on the same processor. We consider mapping issues for a specific architecture in a companion paper[2]. In addition, we introduce communication latency by assuming that the output of every instruction takes l steps to be delivered to its destination. This is consistent with the view that activities are distributed arbitrarily over all the processors. This *finite-processor* latency model has the following attributes:

- Unit time per operation,
- *Bounded number* of operations per step,
- *Fixed communication delay,*
- Unbounded resources.
- *"Fair" (FIFO) scheduling.*

Obviously, for a program whose parallelism profile is less than n throughout, this restriction will not alter its behavior at all. But when it exceeds n, then the effect of execution on n processors may be visualized by drawing a horizontal line at n on the parallelism profile and then "pushing" all the instructions which are above the line to the right and below the line. Figure 9 shows the profile for SIMPLE generated under this model with $n = 1000$, slightly greater than the average parallelism. The length of the critical path is increased from 1,976 to 2,763.

The finite-processor model is somewhat imprecise as a subset of the enabled activities must be chosen in each step and different choices may result in different profiles. We assume a "fair" choice, by which we require that if activity S1 precedes S2 under the ideal model, then it does so under any finite-processor model. In the profiles shown below, ordering among activities which occur in the same step under the ideal model is determined arbitrarily by the way the compiler orders destination arcs. We do not, for example, favor the critical path. To see how the scheduling choice affects the parallelism of a program, we show two different parallelism profiles, based on two scheduling policies: FIFO and LIFO. FIFO scheduling is fair, as defined above. With LIFO scheduling, the most recently enabled instructions are considered before any enabled activities carried over from the previous step. Figure 10 show the results of applying the scheduling orderings to the inner product program graph introduced in Section 2, with only two processors. In general, the performance of the best and worst scheduling choices differ at most by a factor of two [12], and this degree of variance arises in pathological cases.

The discussion thus far has assumed zero communication latency. When latency is introduced in the model, we must be careful in modeling I-structure operations. An I-fetch is a split-phase transactions as shown in Figure 11, so the result of an I-fetch is available at the destination node $2l + 1$ units after the later of the I-fetch and the I-store for the particular elements.

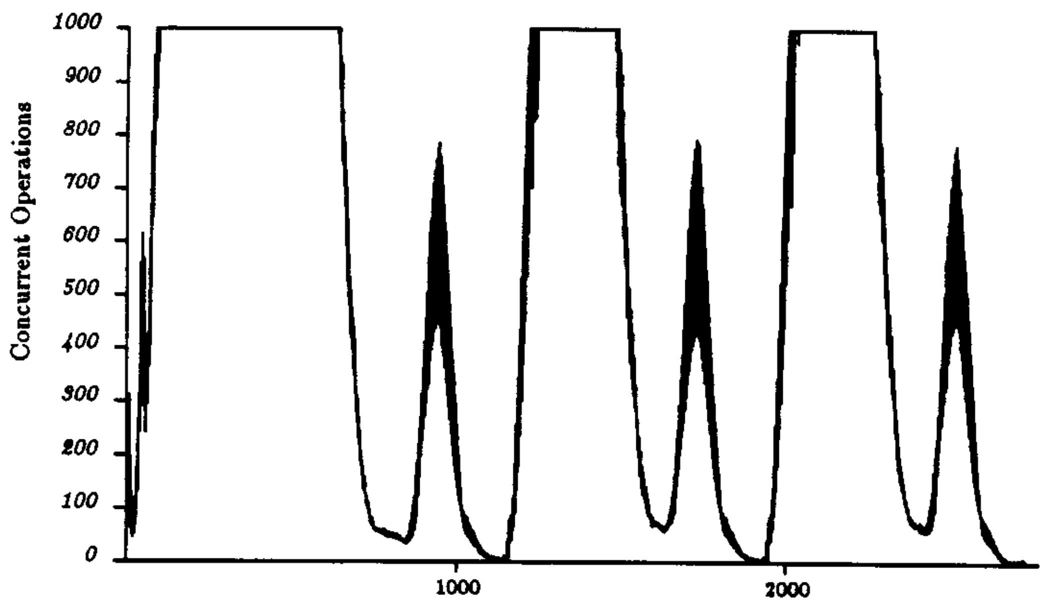

Figure 9: ($p = 1000, l = 0$) Parallelism Profile for SIMPLE (3 iterations, 20 × 20).

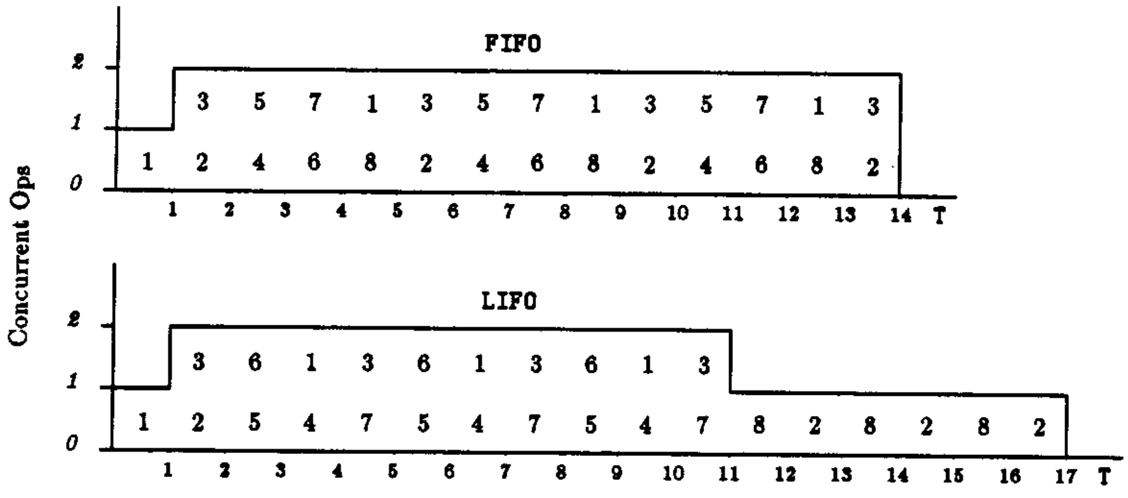

Figure 10: Anomaly in Instruction Scheduling

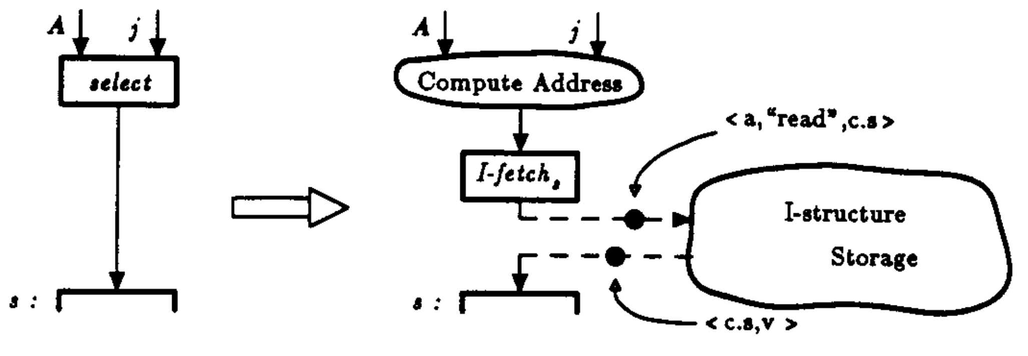

Figure 11: Split-Phase I-Structure Memory Reference

Figure 12 shows two parallelism profiles for 3 iterations of SIMPLE on a 20 × 20 mesh, with at most 100 operations per step. The top profile has latency $l = 0$ while the bottom has latency $l = 10$. Note, the length of the critical path is increased by a factor of 2, not a factor of 10. These profiles suggest that the program has enough parallelism to absorb significant latency on 100 processors.

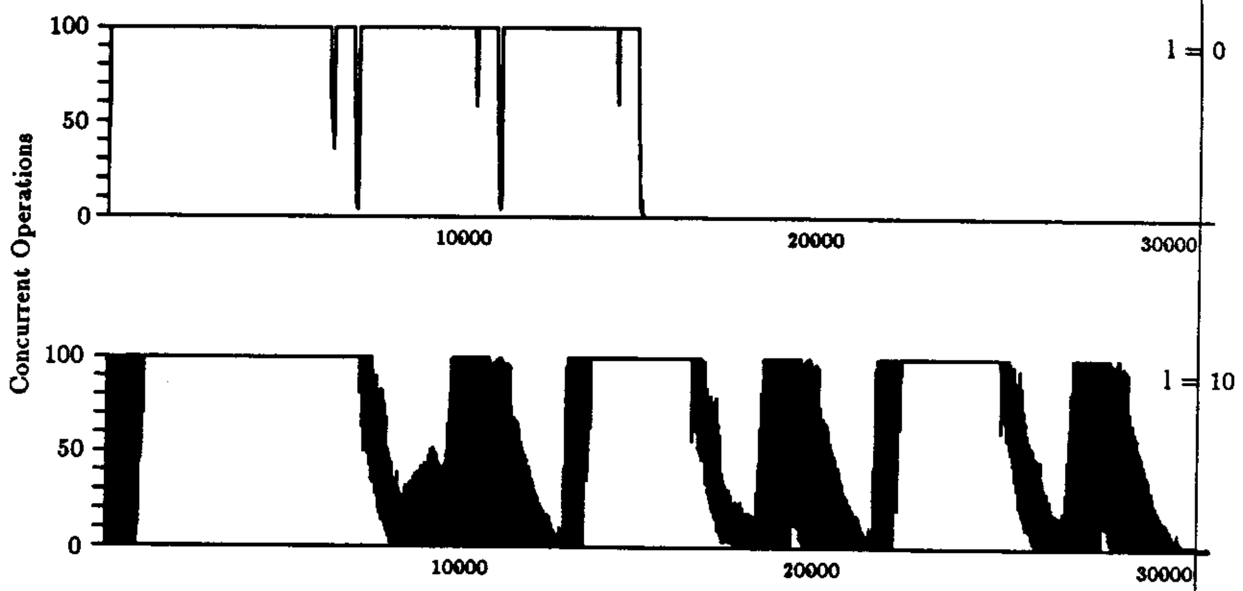

Figure 12: Effect of Latency on Parallelism Profile for Simple (100 processors, 3 iterations, 20 × 20).

4.2 Speedup and Utilization

The amount of parallelism available in a program in the context of the finite-processor model can be expressed in terms of *speedup* and *utilization*, as follows. Let $t(n, l)$ be the number of steps required to execute the program with at most n operations per step and fixed communication latency of l units.

$$speedup(n,l) = \frac{t(1,0)}{t(n,l)} \qquad utilization(n,l) = \frac{t(1,0)}{n \times t(n,l)}$$

$t(1,0)$ is simply the total number of operations executed, *i.e.*,the area under the parallelism profile. These numbers tell us the limits to improved performance imposed by data dependencies in the algorithm itself, modulo influences of instruction scheduling. For example, for 3 iterations of SIMPLE (20×20), *speedup*$(100,0) = 97$, and *utilization*$(100,0) = 97\%$. Thus, even on an idealized machine, *i.e.*,one with instantaneous communication and synchronization, it is not possible to utilize all the processors all of the time.

4.3 Estimating Speedup on Finite Machines from Parallelism Profiles

The ideal parallelism profile provides a means of estimating potential speedup, as an alternative to computing numerous $t(n,l)$ under the finite-processor model. To motivate the approach, we observe that for any τ, if $pp(\tau) \leq n$, we perform all $pp(\tau)$ operations in one step. However, if $pp(\tau) > n$, then we assume it will take the least integer greater than $\frac{pp(\tau)}{n}$ steps to perform $pp(\tau)$ operations. So far the estimate is conservative, because we do not consider that some of the operations from the next step may be performed along with the last few operations of $pp(\tau)$. We must also consider when tokens will have reached their destination nodes. If $l \leq \left\lceil \frac{pp(\tau)}{n} \right\rceil$, then computation of $pp(\tau+1)$ may start immediately after the last operations for $pp(\tau)$, because the earliest results from $pp(\tau)$ have already arrived at their destinations. Otherwise, we must wait l steps from the start of the first n operations of $pp(\tau)$ to begin computing $pp(\tau+1)$. This estimate is optimistic, for consider if all the operations of $pp(\tau+1)$ depended on the last operation performed for $pp(\tau)$. If we assume that dependencies are uniform, we have

$$t(n,l) = \sum_{\tau=1}^{t(\infty,0)} \max(l, \left\lceil \frac{pp(\tau)}{n} \right\rceil), \qquad (1)$$

where $t(\infty,0)$ is the length of the critical path. Figure 13 shows the *speedup* curves derived in this manner for SIMPLE.[4] The points in the figure show the speedup measured under the finite-processor model for various settings of n and l.

Our estimate of $t(n,l)$ tends to be conservative, as explained above. We can partially account for the dynamic instruction scheduling capability of dataflow machines with the following, somewhat optimistic formula.

$$t(n,l) = \sum_{\tau=1}^{t(\infty,0)} \max(1, l, \frac{pp(\tau)}{n}).$$

Finite processor execution gives $t(1000,0) = 2,763$ and $t(100,0) = 15,176$, while our estimation technique shows them to be 2,551 and 15, respectively. Figure 14, shows the speedup and utilization curves for SIMPLE, estimated from the ideal parallelism profile using formula 1.

For realistic architectures the latency is determined by the depth of the processor and storage-controller pipeline, as well as the number of stages in the communications network, which is expected to grow as the log of the number of processors. We have avoided drawing curves where l is a function of n because the appropriate constants are unclear, but suggest that we should consider fairly large

[4]The data for this figure was generated using an earlier version of the compiler (with fewer optimizations).

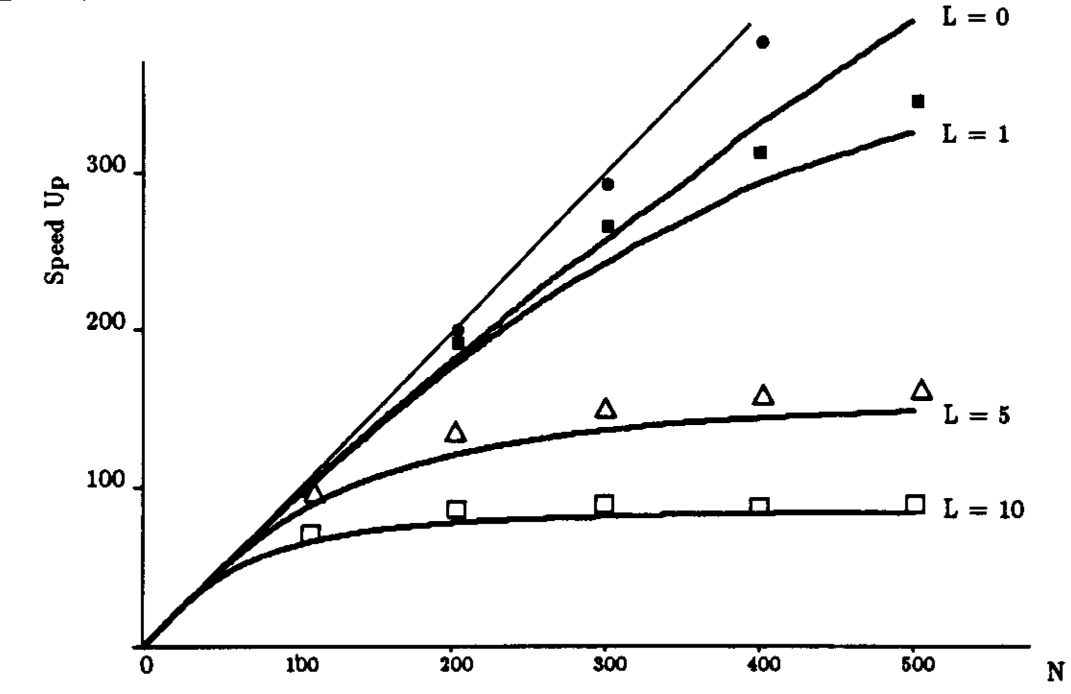

Figure 13: Speedup in the Presence of Latency for Simple (1 iteration, 32 × 32).

latencies. At the same time, since the data comes from an artificially small problem size, we should consider a fairly small number of processors, and extrapolate to realistic problem sizes on more processors.

5 Parallelism and Speedup With Program Partitioning

In characterizing and analyzing the inherent parallelism in programs above we have taken the view that all operations are treated equally. In effect, this models a scenario where any operation can execute on any processor. In practice, there are numerous reasons to require that a collection of operations should execute on the same processor. Advocates of pure dataflow have suggested that this can reduce communication requirements[1], simplify resource management, tag manipulation, and handling of loop invariants[3], and allow for more efficient processor pipelines[17]. Advocates of hybrid von Neuman/dataflow approaches suggest that collections of operators executing on a single processor should be collapsed into a sequential program serving as a kind of "macro" operator[11, 18, 14]. Two natural places to partition programs to form such collections are code-block boundaries *i.e.*, instances of a loop or user-defined function, and iteration boundaries. Although we do not consider it here, one could imagine partitioning the program to minimize the number of arcs crossing the cut, ignoring program structure. Below we consider the amount of parallelism present under the constraint that no more than one operation in each code-block instance (or each iteration) can execute per step.

Two subtleties must be made clear. Let us call a collection of operations which are constrained to execute on one processor a task; it may be a code-block invocation or a single iteration. A task denotes a unit of distribution of work. We allow an unbounded number of concurrent tasks and

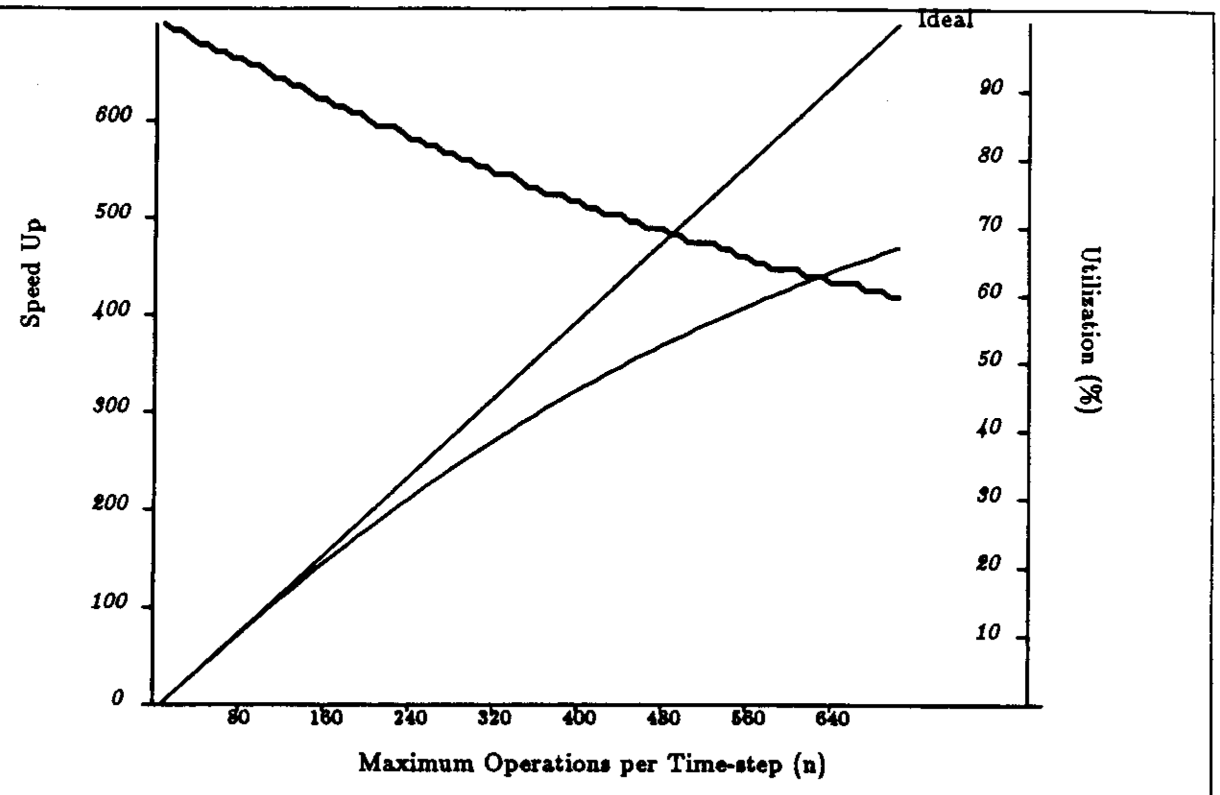

Figure 14: Speedup and Utilization for Simple (3 iterations, 20 × 20).

tasks do not compete for processing resources. Within each task operations are scheduled fairly, as in the finite processor model, based on availability of data; instructions are not statically ordered. Scheduling of operations within different tasks is entirely independent, except as dictated by data dependencies. Any number of tasks can be active concurrently, and in a single step an operation is processed from each task that has any enabled operations. In this way, we are isolating the effects of intratask scheduling constraints without introducing additional effects due to mapping tasks onto a fixed set of processors. The parallelism available under this model can be viewed as an upper bound on that available under hybrid approaches. It should be stressed that within a task, instruction are scheduled according to dataflow firing rules; if a schedule of operations within each task were determined at compile time, the critical paths would probably increase substantially beyond what we show here due to inability to mask memory and communication latency. Below we consider two levels of granularity in partitioning program execution into tasks: code-block level and iteration level.

5.1 Parallelism Under Different Task Granularities

The code-block invocation provides an expedient boundary for partitioning. There are enough such invocations in most programs to yield a sufficient number of tasks for distribution among many processors, and code-blocks form natural boundaries of localized activity. A finer grain partitioning treats each iteration of each loop as a task. We can characterize the inherent parallelism of the program under each task granularity through parallelism profiles generated on an unbounded-processor task-oriented ideal machine, subject to the following specifications:

- Unit time per operation,
- *At most one operation per step per task,*
- *Unbounded number of tasks per step,*
- Zero communication delay,
- Unbounded resources.
- "Fair" (FIFO) scheduling *within each task,*

Figure 15 shows the ideal parallelism profiles of 1 iteration of SIMPLE 32×32 under the instruction level finite-processor model and under the two task granularities [15]. Note these profiles all show operations per step, so the areas under the curves are the same. While the peak parallelism is reduced by factors of 21.5 and 74 from instruction to iteration and to code-block granularity, the more telltale critical paths are lengthened 8.4 and 32 times, respectively, so the average parallelism is 1/8 and 1/32 that under the ideal instruction level model. These factors represent the loss in *potential parallelism* arising from the intra-task scheduling constraints.

5.2 Speedup under different task granularities

In estimating speedup in a finite-processor task-oriented model we place a bound on the number of operations per step, but not on the number of tasks. Since the task-oriented parallelism profiles the number of concurrently enabled operations in each step (subject to intra-task scheduling constraints in addition to data dependencies) speedup can be estimated from these parallelism profiles using Formula 1, taking $l = 0$. Figure 16 shows the estimated speedup from parallelism profiles of SIMPLE 32×32, 50×50, and 64×64. Note that for an $n \times n$ mesh the plateau in the curves of code-block-level partitioning indicates that approximately n-fold parallelism is available.

It is difficult to estimate the effect of latency from the parallelism profile, because different amounts of latency should be charged for inter-task and intra-task communication. To this end, we develop a finite-processor task-oriented ideal machine with the following attributes.

- At most one operation per step per task,
- bounded number of tasks per step,
- Unit time per operation,
- "Fair" (FIFO) scheduling within each task,
- "Fair" (Round-robin) scheduling amongst tasks with enabled operations,
- Zero communication delay for tokens between operations within a task,
- Fixed communication delay for tokens crossing between tasks or to or from I-structures,
- Unbounded resources.

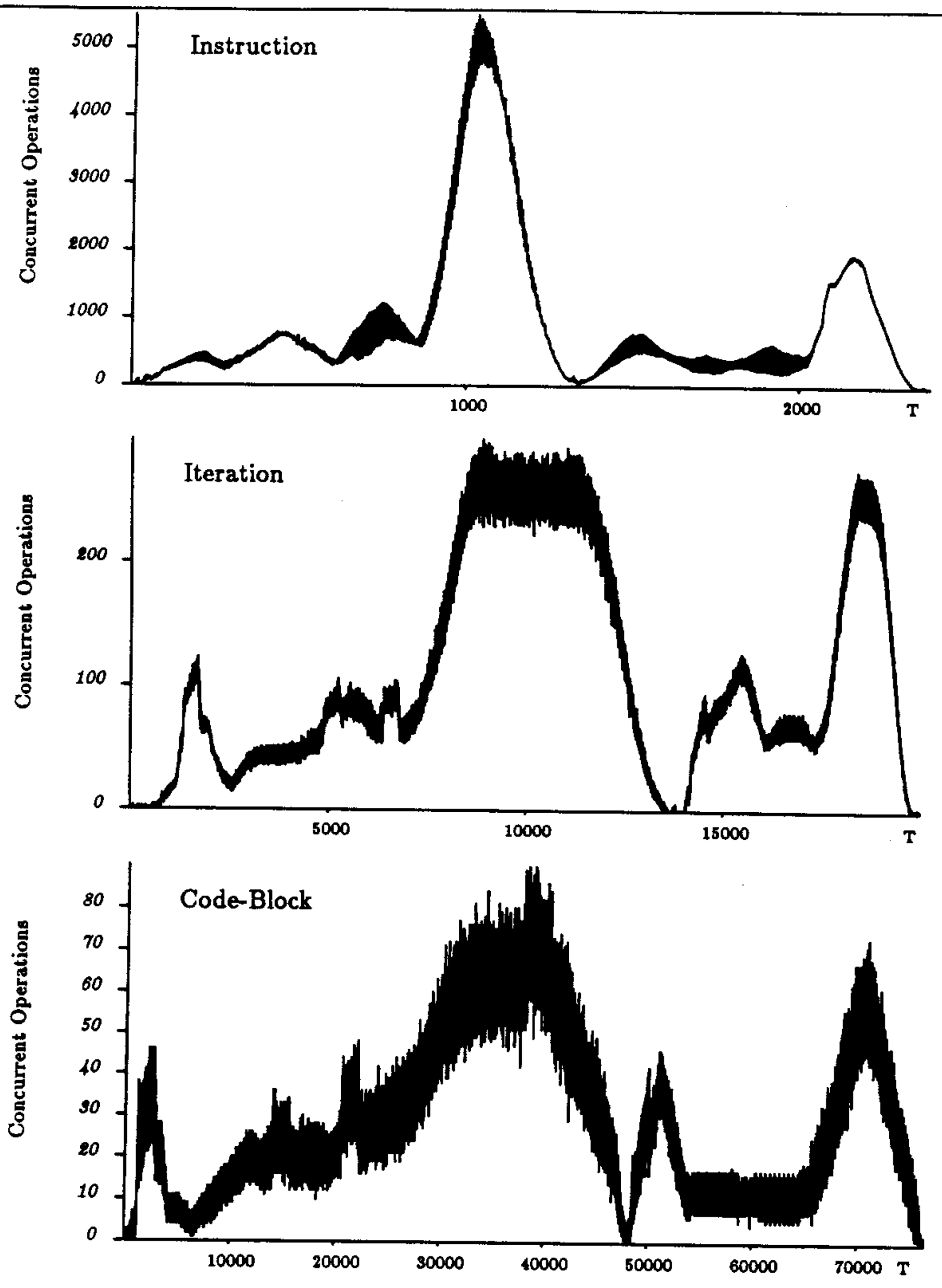

Figure 15: Ideal Parallelism Profiles of SIMPLE(32) under Various Task Granularities.

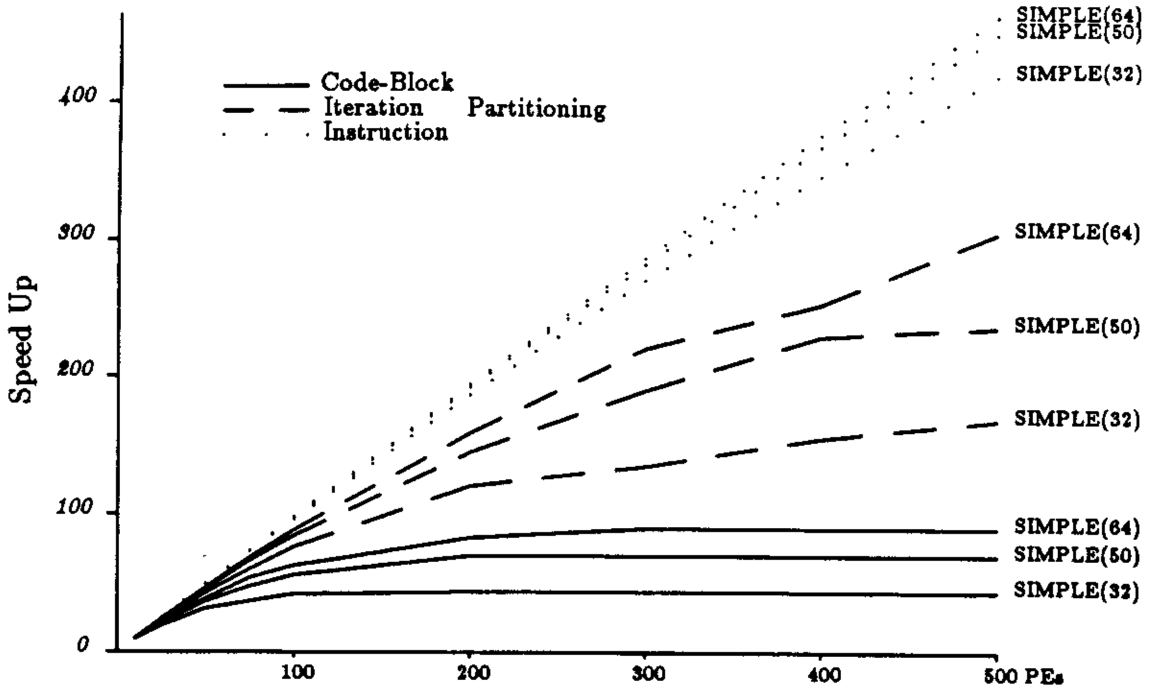

Figure 16: Speedup of SIMPLE(32-64) as Computed from Its Parallelism Profiles.

Under this finite-processor task-oriented model we observe the speedups shown in Figure 17 on SIMPLE 32 × 32 for the three levels of granularity and latency of 0, 10, 20, and 40 units for non-local tokens. The instruction level versions experience this latency on every token. Code-block level versions experience it only in transfer of arguments and results, and in I-structure requests. Still, since I-structure operations constitute roughly 25% of the instruction mix[10], the latency is significant. Iteration level versions experience additional latency on values circulated across iterations. Therefore, we expect that for sufficiently high latency, the reduction in speedup due to latency may be greater than that due to intra-task scheduling constraints. Indeed, at $l = 10$ instruction and iteration level versions cross at roughly 200 processors. At $l = 40$ instruction level falls below code-block level. However, we must note that the crossovers observed here occur at high latency relative to the number of processor, at a large number of processors given the problem size, and ignoring internal latency due to processor pipelining. More realistic scenarios are studied in detail in a companion paper[2].

5.3 Asymptotic Speedup of Programs under Different Partitioning Granularities

While the techniques presented here allow us to analyze applications specified in full detail, the evaluation has been limited to artificially small problem sizes. We would like to extrapolate these results to larger problem sizes, on potentially larger machines. To do this we observe certain trends. Consider once again the effects of partitioning granularity in SIMPLE. We observed in Figure 7 that the behavior of the program is nearly periodic; there is very little useful overlap between

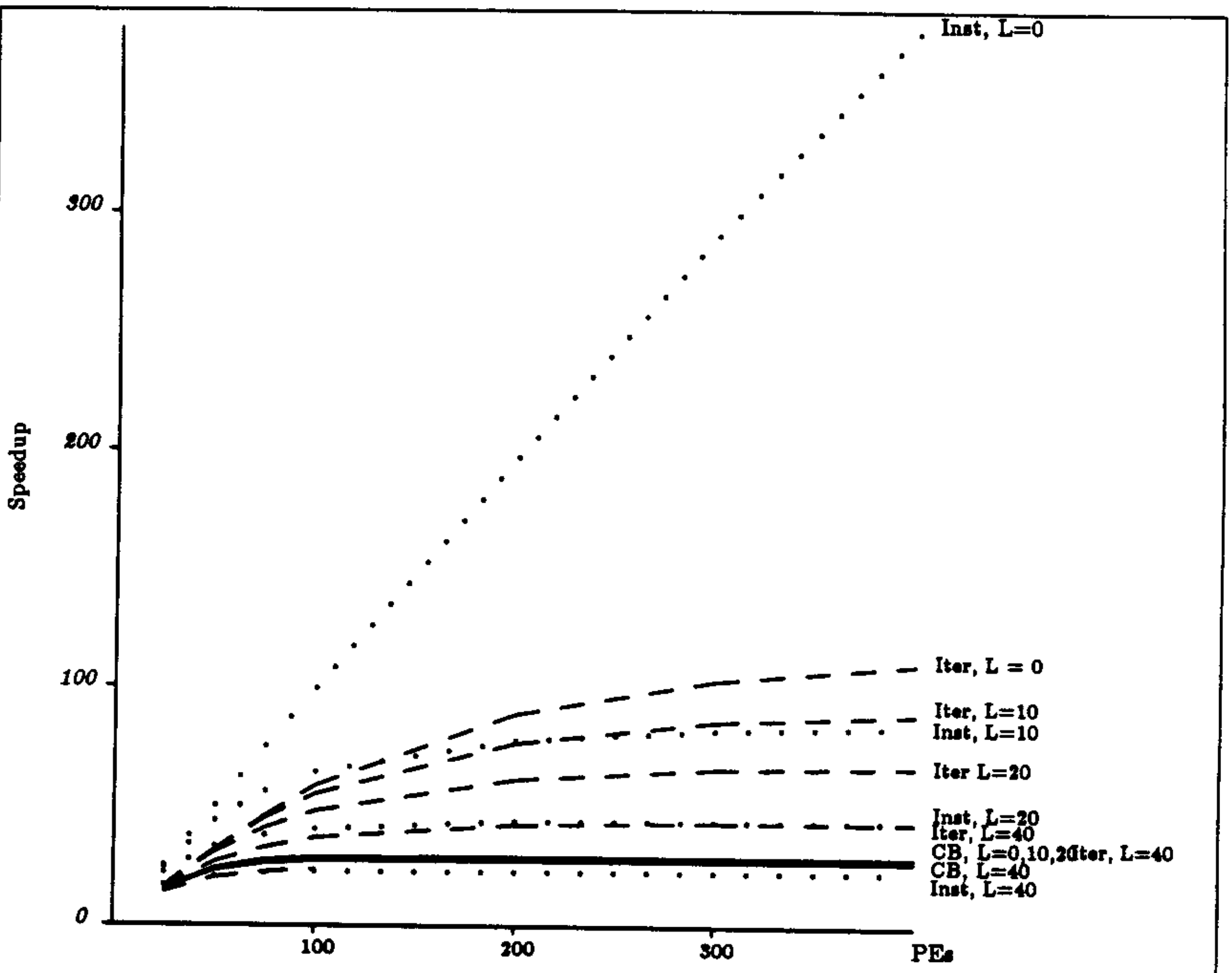

Figure 17: Speedup under various task granularities and latencies for SIMPLE 32 × 32.

iterations.[5] Thus, we can easily extrapolate to more iterations. Changes in the size of the mesh are more interesting.

The following table shows the average parallelism for one iteration of simple with various mesh sizes under the three granularities. Figure 15 represents the middle column. Looking across the columns we see that the average parallelism changes with problem size.

Mode	10 × 10	16 × 16	32 × 32	50 × 50	64 × 64
Instruction	178	361	922	1584	2108
Iteration	19	40	110	199	270
Code-block	8	14	29	46	59

To get a better idea of the trends, we consider the ratio of average parallelism, which effectively normalizes the problem size, giving the following table.

Ratio	10 × 10	16 × 16	32 × 32	50 × 50	64 × 64
Instruction / Code-block	22.3	27.8	32.9	35.2	35.7
Instruction / Iteration	9.9	9.0	8.4	8.0	7.8
Iteration / Code-block	2.25	3.1	3.9	4.4	4.6

Here the trends are more apparent. The loss of parallelism under code-block granularity becomes significant as the problem size increases. However, the loss of parallelism under iteration level decreases. This reflects inner loops, which tend to be small, becoming more dominant. Indeed, we see that the ratio of average parallelism in iteration versus code block granularity grows faster than instruction versus code-block granularity. This suggests that iteration level granularity is even more attractive for large problem sizes. These results are highly dependent on problem structure, however, and should consider carefully as in [10].

6 Conclusion

We have presented a method for quantifying the parallelism in *real* programs developed in the context of a dataflow model, but generally useful. It allows programs to be studied in full detail, without biasing their behavior by implementation constraints. This allows us to draw a clear distinction between the parallelism inherent in a program and the speedup achieved under a specific implementation. We have presented two methods for deriving upper-bounds on potential speedup and presented data on how speedup is effected by latency and intratask scheduling constraints. Studies using the techniques presented here have substantiated our belief that large programs employing traditional algorithms exhibit "sufficient" parallelism when coded in a dataflow language for machines of reasonable size. Moreover, this work brings us closer to answering questions of the form, "Does this program have sufficient parallelism for a machine of n processors?", considering the characteristics of the program and the characteristics of the machine. Certainly, we would like to verify these predictions on real machines. A companion paper[2] examines the observed speedup on a detailed simulation of the MIT Tagged-Token Dataflow Architecture as a first step.

[5] A companion paper shows that there is, in fact, a hazardous kind of overlap between iterations, but that this can be removed.

References

[1] Arvind. Decomposing a Program for Multiple Processor System. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 7–14, August 1980.

[2] Arvind, Stephen A. Brobst, and Gino K. Maa. *Evaluation of the MIT Tagged-Token Datflow Architecture.* Technical Report Computation Structures Group Memo 278, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. (Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

[3] Arvind, David E. Culler, Robert A. Iannucci, Vinod Kathail, Keshav Pingali, and Robert E. Thomas. *The Tagged Token Dataflow Architecture.* Technical Report FLA, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1983. Revised October, 1984.

[4] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987.

[5] Arvind and K. P. Gostelow. The U-Interpreter. *COMPUTER*, 15(2), Feburary 1982.

[6] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259)*, Springer-Verlag, June 15-19 1987.

[7] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The SIMPLE Code.* Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[8] David E. Culler and Arvind. *Resource Requirements of Dataflow Programs.* Technical Report Computation Structures Group Memo 278, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. (Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

[9] Jack B. Dennis. First Version of a Data Flow Procedure Language. In G. Goos and J. Hartmanis, editors, *Proc. Programming Symposium, Paris 1974 (Lecture Notes in Computer Science 19, Springer Verlag)*, Spinger-Verlag, New York, 1974. (Revised: MAC TM61, May 1975, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[10] Kattamuri Ekanadham, Arvind, and David E. Culler. *The Price of Parallelism.* Technical Report Computation Structures Group Memo 278, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. (Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

[11] J. L. Gaudiot and M. D. Ercegovac. Performance Evaluation of a Simulated Dataflow Computer with Low Resolution Actors. In *Journal of Parallel and Distributed Computing*, Academic Press, December 1987.

[12] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[13] W. Daniel Hillis and Guy Lewis Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[14] Robert A. Iannucci. *A Dataflow/von Neumann Hybrid Architcture.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, January 1988 (expected).

[15] Gino K. Maa. *Code-Mapping Policies for the TTDA.* Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, December 1987 (expected).

[16] Rishiyur Sivaswami Nikhil. *Id World Reference Manual.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[17] Gregory M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, June 1988 (expected).

[18] Requa and Mcgraw. Piece-wise dataflow. ??

[19] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture.* Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).