LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

# Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs

Computation Structures Group Memo 279
March 1988
Revised June 1988

**Arvind**
**David E. Culler**
**Gino K. Maa**

To appear in the *Journal of Supercomputing Applications* and the *Proceedings of Supercomputing 88*, Orlando, FL, November 14-18, 1988.

This report describes research done at the Laboratory for Computer Science at 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. Mr. Culler received support from Unisys as a Burroughs Scholar.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

# Assessing the Benefits of Fine-grain Parallelism in Dataflow Programs[1]

Arvind
David E. Culler
Gino K. Maa

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

## Abstract

A method for assessing the benefits of fine-grain parallelism in "real" programs is presented. The method is based on *parallelism profiles* and *speedup curves* derived by executing dataflow graphs on an interpreter under progressively more realistic assumptions about processor resources and communication costs. It is shown that programs, even using traditional algorithms, exhibit ample parallelism when parallelism is exposed at all levels, *i.e.*, within expressions, across nested loops and function calls, and in producer-consumer relationships on individual elements of data structures. Since we only consider dataflow graphs compiled from the high-level language Id, the bias introduced by the language and the compiler is examined. A method of estimating speedup through analysis of the ideal parallelism profile is developed, avoiding repeated execution of programs. It is shown that fine-grain parallelism can be used to mask large, unpredictable memory latency and synchronization waits in architectures employing dataflow instruction execution mechanisms. Finally, the effects of grouping portions of dataflow programs, such as function invocations or loop iterations, and requiring that the operators in a group execute on a single processor, are explored.

## 1 Introduction

The high-performance computing community has engaged in a long-standing debate on the proper "granularity of parallelism" for multiprocessors [24]. Coming from a mind-set of migrating sequential programs to complexes of sequential machines, one tends to view sequential execution as fast compared to the time required for communication and synchronization. It is observed that as the logical partitions of a program are made smaller the amount of external communication increases and this can offset the advantage of parallel execution. While this view is basically correct, the mind-set encourages one to consider only the sorts of parallelism these machines can exploit, and this can lead to the dubious conclusion that the "useful" parallelism in programs is small. We come into the debate with the opposite bias — a dataflow model where *any* two operators can execute

[1]This research was done at the MIT Laboratory for Computer Science. Funding for this project 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.

in parallel, unless one actually provides data directly or indirectly to the other, and where no fundamental distinction is made between inter- and intra-processor communication. From this view point, it makes sense to ask how much parallelism is inherent in a program; can machines be built to exploit this kind of parallelism; and what advantages may such machines hold over currently available machines. If, indeed, a large class of programs have substantial fine-grain parallelism and the machines that exploit this parallelism are cost-effective, we can expect tremendous performance improvements in general-purpose computing.

In this paper, we focus on the first question and develop a methodology for studying inherent parallelism in programs. Though our methodology is developed in the context of dataflow, we believe it is applicable to other types of systems as well. Dataflow program graphs provide a precise description of parallel computation[12]. Arcs indicate data dependence between operations, and operators not connected by paths in the graph represent opportunities for parallel execution. In Section 2, 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 length, average parallelism, and potential speedup can be deduced.[2]

Program graphs are generated by compiling Id, a high-level declarative language [20, 7] which holds the potential of making a dent in the cost of producing application software. In Section 3, we examine the bias that this introduces and restrict our attention to problems which are comfortably expressed both in Id and traditional languages. In Section 4, we offer experimental evidence for our primary claim: *parallelism is pervasive, even in programs employing traditional algorithms*, provided that parallelism is exposed at all levels, not only in loops and function calls but even in producer/consumer relationships which may require synchronization on individual elements of data structures. We show how failing to exploit certain kinds of parallelism can significantly increase execution time. We expect that only architectures geared to provide synchronization and communication efficiently can exploit the fine-grain parallelism in dataflow graphs.

In Section 5, we consider the behavior of programs under the constraint that at most a certain number of operations can be performed simultaneously. This gives us a way to compute the potential speedup curve of a program for a finite processor machine. We then consider the impact of communication latency, showing that as the latency increases more parallelism is required to achieve a given speedup. Another interpretation of this data is that fine-grain parallelism can be used to mask large, unpredictable communication latency and synchronization waits, which will occur in any parallel architecture. To avoid the repeated executions that are needed to generate the parallelism profiles of a program, a method for *estimating* the speedup from a single parallelism profile is also given.

Finally, we introduce program partitioning at various levels of granularity into the execution model. Since the operators in a partition are not allowed to execute in parallel, the available parallelism is reduced; Section 6 examines its effects on potential speedup. We also consider the more realistic scenario in which external communication is expensive, and examine the trade-offs between the granularity and the speedup.

Throughout this paper we assume that the reader has some familiarity with dataflow; a good introduction to the MIT Tagged-Token Dataflow Architecture may be found in [5].

[2] Elsewhere [1] we have presented a detailed account of the fraction of operators that represent the overhead or the cost of parallel execution. Resource requirements of parallel programs are discussed in [11]. Taken together, these papers provide a better characterization of the "useful" parallelism in dataflow programs than this paper alone does.

## 2 Parallelism Profiles under Ideal Execution

Dataflow is an asynchronous, time-independent model of computation; nonetheless, in order to characterize the parallelism in dataflow programs, we consider a synchronous execution model with unit time per operation. Timing assumptions are purely for quantification; we do not exploit any aspect of synchronous execution in dataflow program graphs.

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:

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

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

We illustrate the method for generating parallelism profiles through an example. Figure 1 shows a simplified program graph which computes the inner products of two vectors, `A` and `B`, of size `n`. Initially, a token for **sum**, with value zero, is sent to the left switch, and tokens for `i`, with value one, are sent to the right switch and the $\leq$ predicate. The value `n` and the descriptors for the two vectors are *loop invariants*[5], 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 that are instantly available at the control inputs of the two switches. Instructions 2 and 3 fire in the second step and produce 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. The 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 that, in step 5 of the parallelism profile, the second iteration has begun while the first is still active. Execution continues in this manner until step 14 when a token on the FALSE side of the switch in instruction 2 is produced. The pattern in steps 4 to 6 covers all 8 instructions and repeats for every iteration. Note, an operator is executed at the step corresponding to the later of the arrival times of its input tokens.

The step beyond which $pp(t)$ is uniformly zero is called the *critical path length*, and is denoted by $T_\infty$. Thus, $T_\infty$ 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 $T_1$. The ratio of these is the *average parallelism*, $\overline{P} = T_1/T_\infty$. The parallelism profile of the inner product of length 3 is shown in Figure 1. It can be seen from it 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; it may be possible to achieve greater parallelism with a better algorithm, but that would be described by a different graph.

## 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[25]. Id is a functional language, extended with I-structures[6] to provide efficient array manipulation. A broad class of algorithms

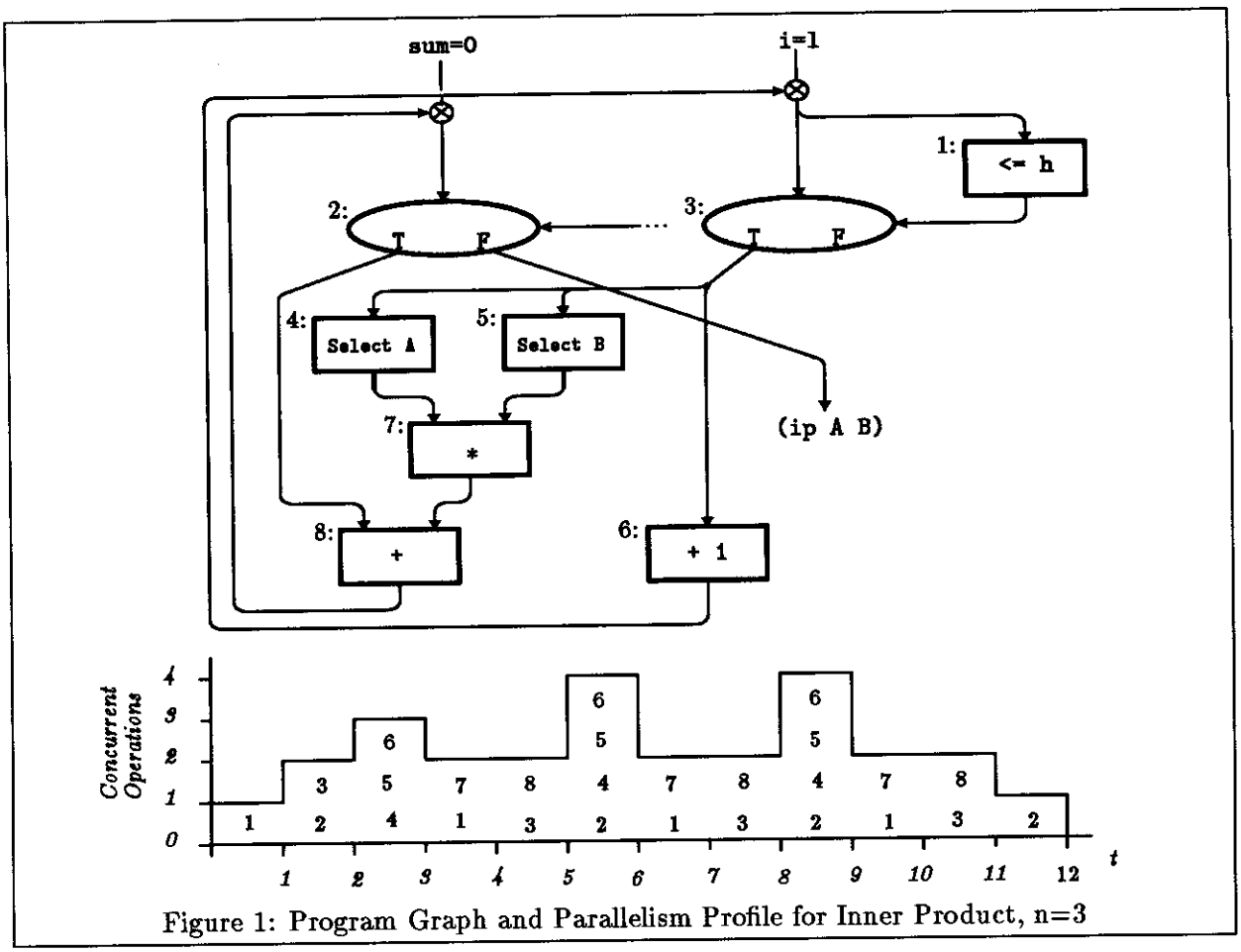


Figure 1: Program Graph and Parallelism Profile for Inner Product, n=3

are easily expressed in Id, although we recognize that it is difficult, perhaps impossible, to express efficient algorithms for certain types of problems in the current version of Id. 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 schemas and rules of composition which are described in [5, 25]. Very briefly, arithmetic expressions are translated into acyclic graphs. Blocks, which give names to expressions, are treated as "wiring diagrams" to compose graphs. Conditional expressions are constructed using switch operators to steer values to the appropriate arm based on a predicate. Iteration is captured by a loop schema, which permits arbitrary overlap of iterations unless constrained by data dependencies. User defined functions and loops are compiled into code-blocks, which are invoked by an apply schema, permitting arbitrary recursion. (Values which are arguments to an invocation of a loop but are constant over all iterations are not circulated; instead, they are explicitly stored in a constant area (see [5]) as part of the loop preamble.) The graphs generated by the Id compiler are determinate and self-cleaning. Furthermore, graphs are embellished such that each code-block receives a "trigger" to enable operators without any data inputs, *i.e.,* operators all of whose inputs are literals or loop constants. To facilitate termination detection, all operators that don't have a natural output, *e.g.,* the store instruction, are made to produce a "signal" indicating that they have consumed their inputs[25].

The graph generated by the Id compiler for `ip`, the inner-product program, contains 31 instructions. Most of these are for loop set-up and termination detection, and are 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 [21], 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, the 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, discussed further in [11]. 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[1] while preserving determinacy.

## 3.2 Compiler Optimizations

The Id compiler performs most of the traditional optimizations, including constant propagation, common subexpression elimination, and code hoisting, plus optimizations which are particular to functional languages, such as tuple elision and arity analysis. A powerful function *inlining* capability is provided to reduce the overhead of function application. The traditional optimizations are very effective on dataflow program graphs, because of the clean semantics and structure of dataflow

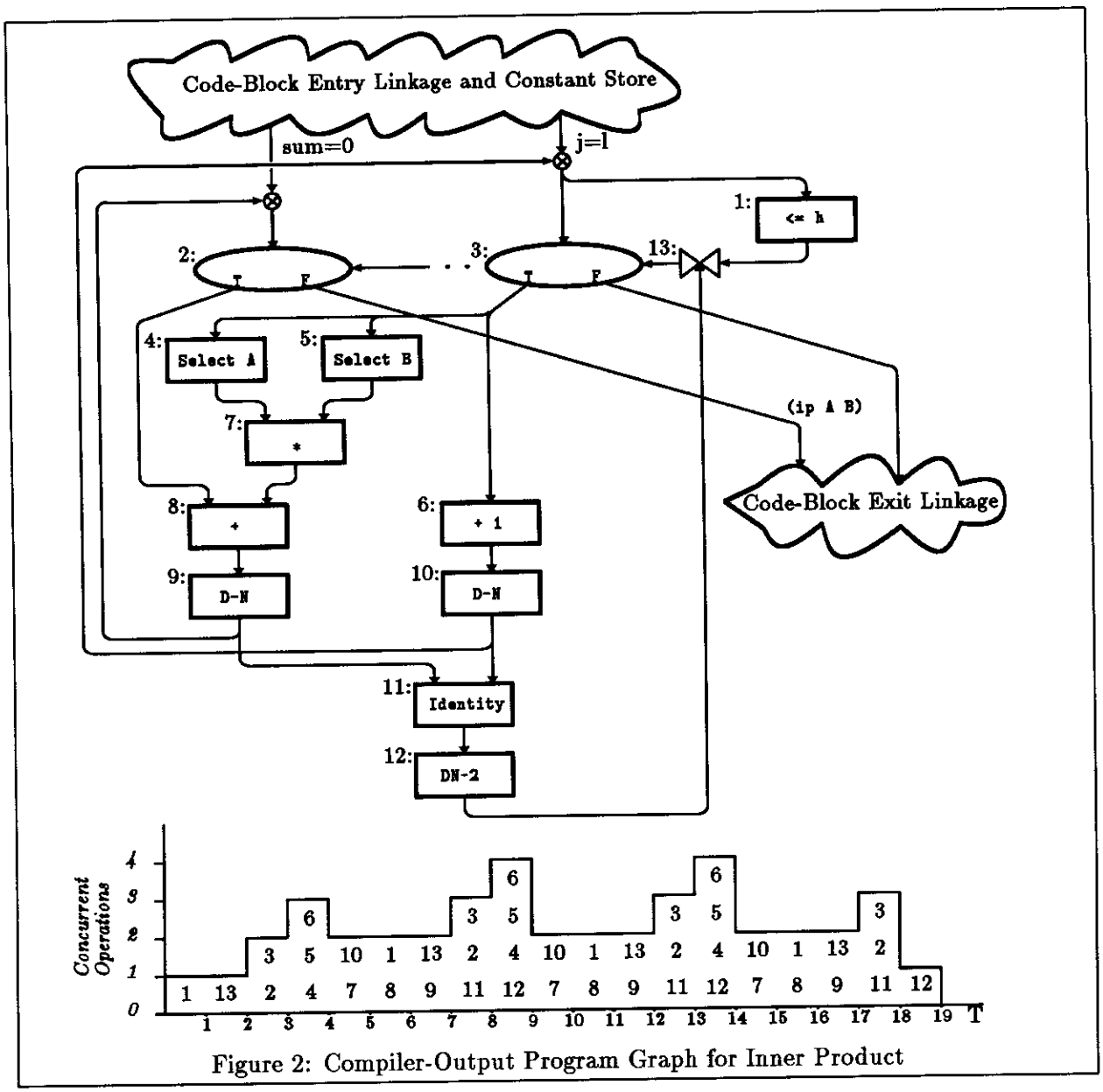


Figure 2: Compiler-Output Program Graph for Inner Product

graphs. Optimizations such as loop fusion and unrolling, and incorporation of algebraic boolean and relational identities are planned for the future.

Note that in the inner-product example, the multiplications could execute in any arbitrary order and the sum would still be formed in sequence. In particular, if the `select` operators incur a delay in providing data to the multiply because of communication latency, the loop will unfold and multiple outstanding fetches will be issued, masking the latency. The critical path can be shortened if the compiler can recognize associative operations, reassociating them into balanced trees. For this problem, the programmer could also have used a recursive formulation to do a binary tree summation to achieve the same effect. It is also possible to describe dataflow graphs which combine arbitrary partial products, in whatever order they happened to be produced, but this will require a non-deterministic operator. Intentionally, Id 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.

# 4 Fine-grain Parallelism in Programs

Generally, a given machine can exploit only certain kinds of parallelism. Most machines exploit parallelism within the instruction cycle through instruction prefetch. Many machines exploit local parallelism in an instruction sequence by using pipelined execution units. Vector processors [23] exploit parallelism in simple, inner-most loops. Long-instruction-word machines [8, 13] exploit fine-grain parallelism within many kinds of loops and possibly in instruction sequences, but generally do not consider executing completely unrelated loops together. Multicomputers generally exploit parallelism between large tasks, but little within a task. Only a few machines, the Sigma-1 [15] and HEP [17] being the most notable examples, have the capability to support parallelism between producers and consumers of data structures. The dataflow model does not draw a distinction between different kinds of parallelism; independent operators within an expression are treated in the same way as the operators within a vectorizable loop or those in large unrelated tasks. In this section we show how this results in ample available parallelism.

## 4.1 Non-strict Data structures

Generally, imperative languages compromise parallelism in manipulation of data structures by requiring a barrier between the producer of a structure and its consumers. Functional languages with "strict" semantics exhibit a similar shortcoming because data structures are treated as a single value, and operations which access particular elements of the structure cannot execute until the entire structure is defined [12]. This prevents asynchronous production and consumption of structures, much as does barrier synchronization. Our model avoids this limitation through I-structures, 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 that is not present is deferred until the element is written, and processed immediately thereafter. To ensure determinacy, an element can only be written once. 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 the 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 = {l,h = bounds a;
                C = ARRAY (l,h) ;
                {FOR j FROM l TO h 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. I-structure operations introduce a minor complication in computing parallelism profiles. The operator which receives the result of a `select` operation does so one unit after either the time of the `select` or the time of the corresponding store, whichever comes later.

Figure 3 shows the parallelism profiles for `vsum`, `ip` and `ip_vsum` respectively with $n = 10$. The critical path is 64 for `vsum` and the total operations 158. For `ip` they are 60 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 $64 + 60$, is only 77, even though the instruction count is 500. The last number is greater than 469 ($= 153 + 2 \times 158$) because of the three procedure calls. Figure 3 also introduces our notation for ideal profiles: $p = \infty$, unbounded operations per step, and $l = 0$, zero latency. More realistic models, to be introduced later, are indicated by the particular values of $p$ and $l$.

Certainly, loops as simple as these could be made to work in tandem under synchronous execution, or by loop fusion transformations. However, such approaches break down when the producer and consumer are large, complex processes with irregular accessing patterns and varying computational requirements, operating asynchronously under the influence of unpredictable communication latency due to network contention and other factors. I-structures are still effective in this general setting.

## 4.2 Parallelism in Nested Loops

Composition of control structures does not inhibit parallelism in dataflow programs. In contrast, some approaches exploit parallelism only in outer-most or inner-most loops, or only within a procedure, or only in spawning a procedure. Id allows loops to unfold and functions to be spawned, irrespective of the nesting of control structures, constrained only by data-dependencies.

Figure 4 gives the parallelism profile for matrix multiplication of two $16 \times 16$ matrices by 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 8,192 are floating point operations. The average parallelism is 249 operations per step. The bell-shape of the profile arises because loop unfolding causes inner loops to be staggered slightly, and the summation in

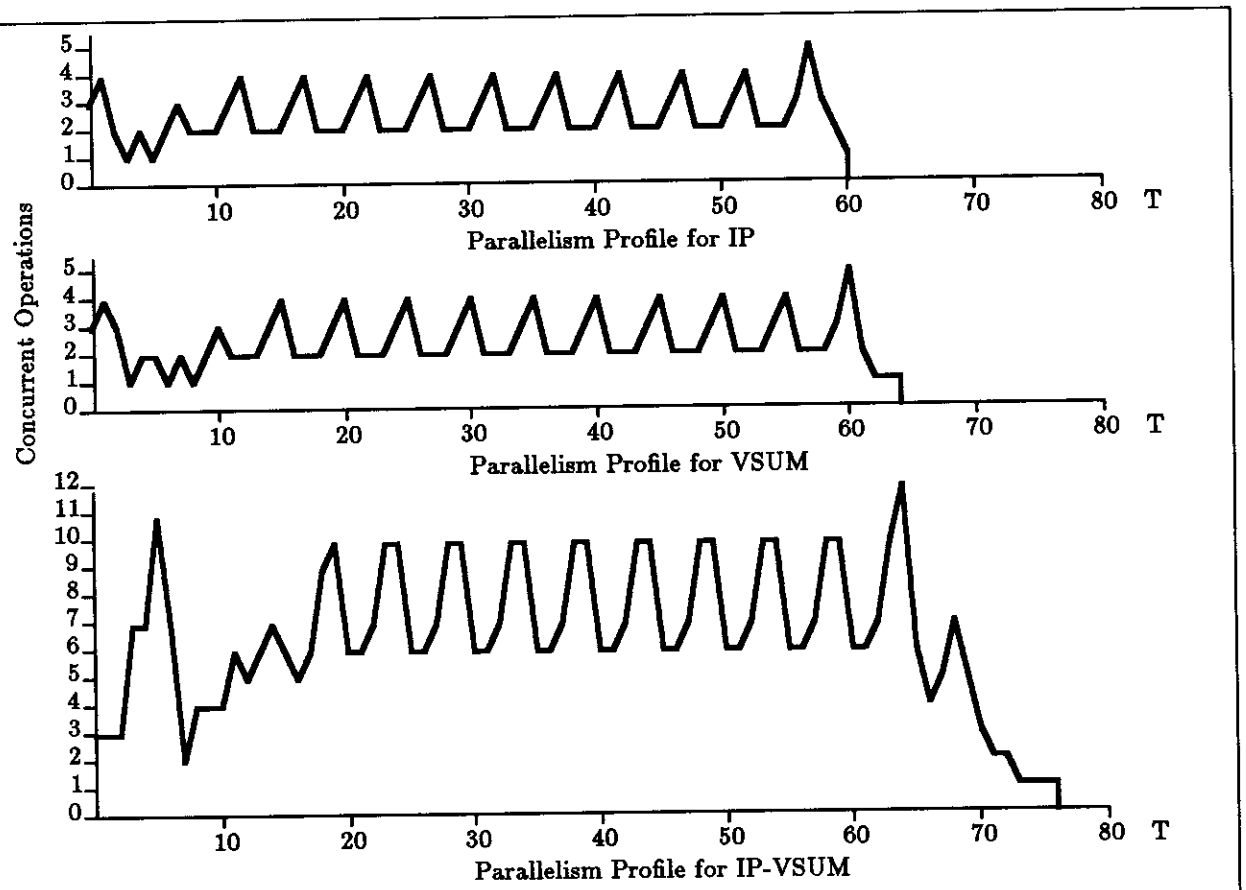


Figure 3: $(p = \infty, l = 0)$ Parallelism Profiles for Vector Sum, Inner Product and their composition (size 10).

the inner-most loop is done sequentially. The rapid oscillations reflect the cumulative effect of the variations in parallelism in the inner-most loop. We could spawn the inner loops in a tree, and perform the summations as a binary tree to reduce the length of the critical path further, but it is evident that substantial parallelism is present even in the traditional algorithm.

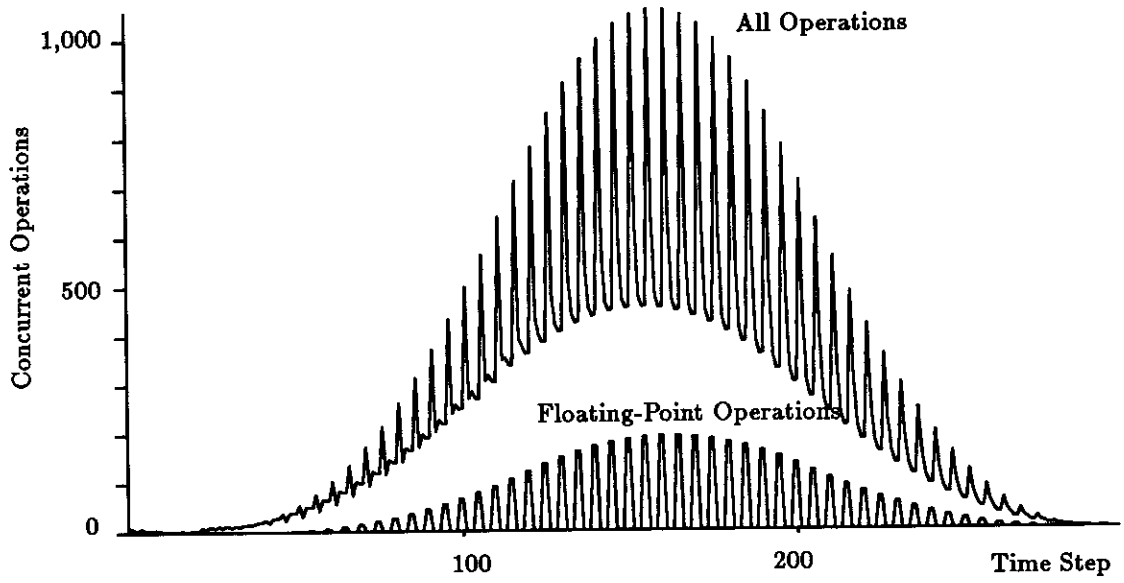


Figure 4: ($p = \infty, l = 0$) Parallelism Profile for Matrix Multiply ($16 \times 16$)

LU decomposition is the central component of many equation solvers. It involves nesting of loops and conditionals. An Id program for LU decomposition with partial pivoting is given below.

```
Def decompose A n =
 { D = matrix ((1,n),(1,n));
   B = {For k From 1 To n-1 Do
          r = find_pivot A;

          {For j From k To n Do
             D[k,j] = A[r,j] };

          {For i From k+1 To n Do
              D[i,k] = -A[i,k] / A[r,k] };

          Next A = { A' = matrix ((k+1,n),(k+1,n));
                     {For i From k+1 To n Do
                        {For j From k+1 To n Do
                           A'[i,j] = D[i,k]*D[k,j] + A[(If i==r Then k Else i),j] }}
                   In
                     A' }
        Finally A};
   D[n,n] = B[n,n];
 In
   D};
```

Figure 5 shows the parallelism profile for a 16 x 16 LU decomposition (critical path = 1312 , total operations = 35,552). The descending stair-step behavior coincides with our intuition that after each pivot the remaining portion of the matrix has one less row and one less column.

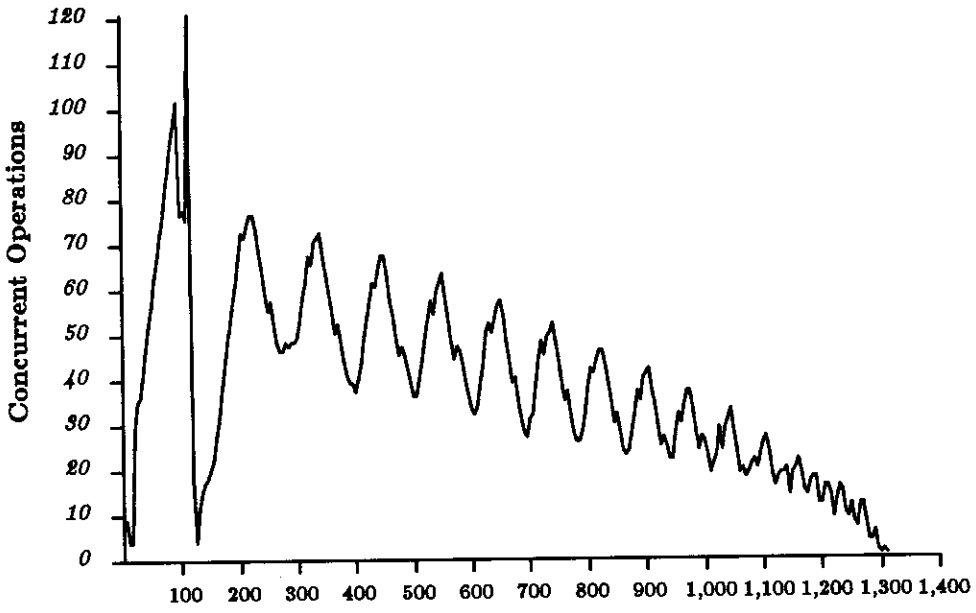


Figure 5: ($p = \infty, l = 0$) Parallelism Profile for LU Decomposition with Partial Pivoting (16×16)

### 4.3 Recursive Data Structures and Loops

One normally thinks of producer/consumer relationships as being between two distinct portions of a program, but recursive data structures can create such a situation between successive iterations of a single loop. The example below, based on common relaxation methods, shows such a situation. In addition, this example demonstrates that a program may have ample parallelism, even though every loop appears to be a linear recurrence in isolation. The only way to expose this parallelism is to allow all the loops to unfold together.

```
DEF wave previous_m =
  {m = matrix ((1,n),(1,n));
   m[1,1] = previous_m[1,1];
    {FOR i FROM 2 TO n DO
      m[1,i] = previous_m[1,i];   % Copy left and upper edges
      m[i,1] = previous_m[i,1];
                                  % Fill in the rest by averaging
       {FOR j FROM 2 TO n DO      % with left and upper neighbors
         m[i,j] = (m[i-1,j] + m[i,j-1] + m[i-1,j-1] +
                                        previous_m[i,j]) / 4}
     }
   IN m};

DEF multiwave mesh k = {FOR i FROM 1 TO k DO
```

```
        NEXT mesh = wave mesh;
    FINALLY mesh} ;
```

Computations along the diagonal in **wave** can be executed in parallel, so a single invocation of **wavefront** exhibits a profile which increases linearly as the diagonal grows to $\mathcal{O}$(n) and then decreases as the diagonal shrinks. Multiple invocations of **wave** in **multiwave** generate a plane of parallel activity, giving the profile with a single peak shown in Figure 6. The other profile in Figure 6 shows the behavior when parallelism is exposed within **wave**, but barriers are placed between iterations of **multiwave**, excluding producer/consumer parallelism. To achieve even this level of parallelism under more traditional approaches, it is necessary to transform **wave** to have a much more complicated control structure; an outer loop traverses the diagonal of the matrix sequentially and a parallel inner loop computes the elements on the perpendicular diagonal [18].

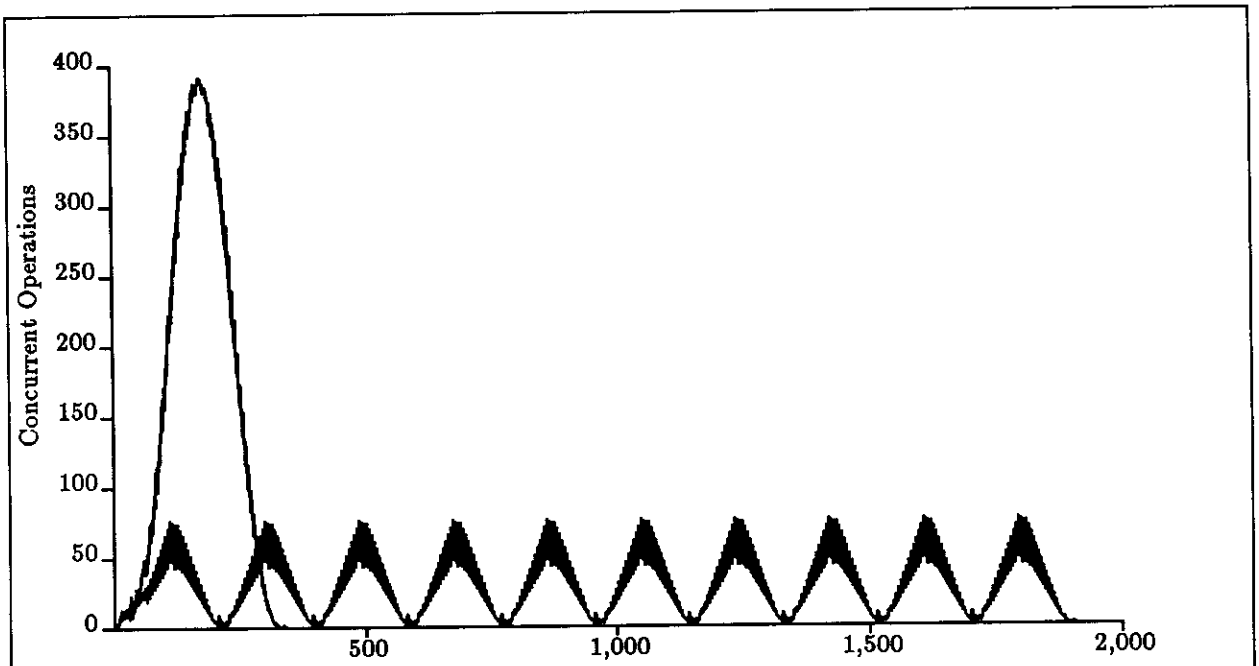


Figure 6: ($p = \infty, l = 0$) Parallelism Profiles for Multiwave, 10 iterations on a 16 x 16 mesh

## 4.4 Large Application Kernels: PIC and SIMPLE

We now discuss the parallelism profiles of the kernels of two large applications. These kernel codes are of the order of 1000 lines of Id, and are complex enough to make static analysis of parallelism a very difficult task. However, it is often possible to explain even complex parallelism profiles after they have been generated, provided one has a good understanding of the application.

Figure 7 shows the parallelism profile for an 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 x 8 mesh); realistic problems use 1 million particles and 100,000 cells, approximately. For the problem in Figure 7, the number of operations executed is 1,573,733 and the critical path length is 3,399.

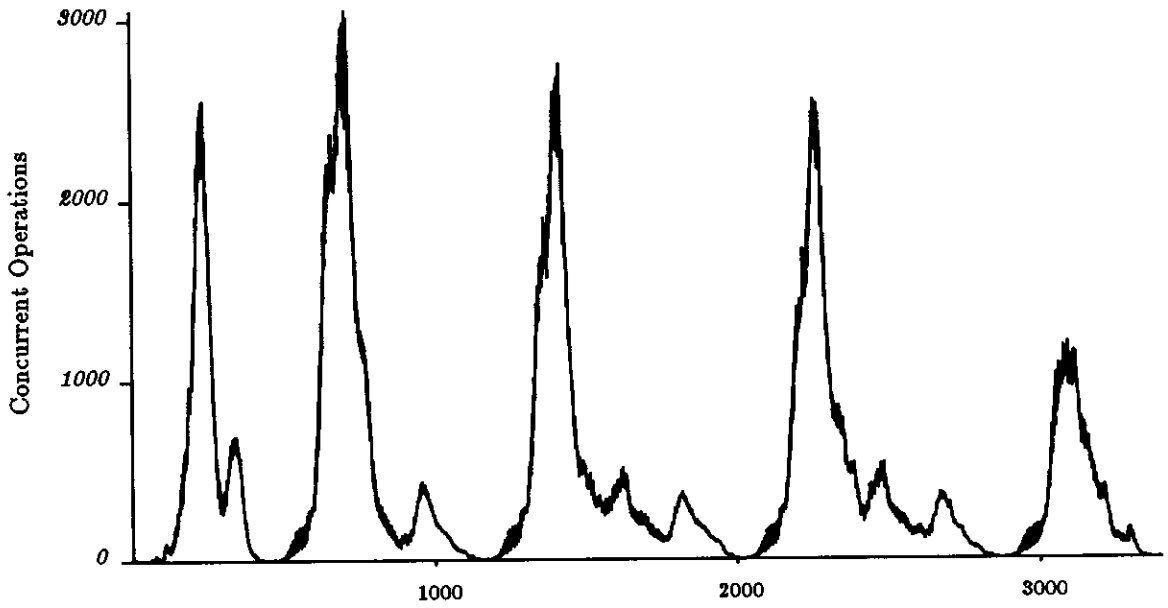

Figure 7: ($p = \infty, l = 0$) Parallelism Profile for Particle-In-Cell (4 steps, $8 \times 8$ cells, 640 particles)

In PIC, a large amount of parallelism is generated while computing the potential and solving Laplace's equation. This particular program uses a multigrid method to solve Laplace's equation, which generates the first peak in the parallelism 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 new time step. Once the time step is determined, the computation of new particle positions begins, generating substantial parallel activity. We think, however, that the computation of the new charge density overlaps with the particle push, while the new time step is being determined. After the second iteration, the behavior is periodic. It is possible for us to verify the above analysis by tracing operators belonging to a particular function call.

Figure 8 shows the parallelism profile of the SIMPLE code, a hydrodynamics and heat flow code that has been studied extensively both analytically [9] and by experimentation. A detailed discussion of the program appears elsewhere [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 iteration's 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 9. (Critical path = 1082, instruction count = $1,446,478$.)

Hidden in this profile is substantial producer/consumer parallelism, possible only with element by element synchronization. A total of thirteen matrices are created during each iteration, in eight distinct phases of the computation. If barriers are introduced between producers and consumers, the critical path increases by more than a factor of three.

Parallelism profiles accurately capture the inherent parallelism in programs, given the specific

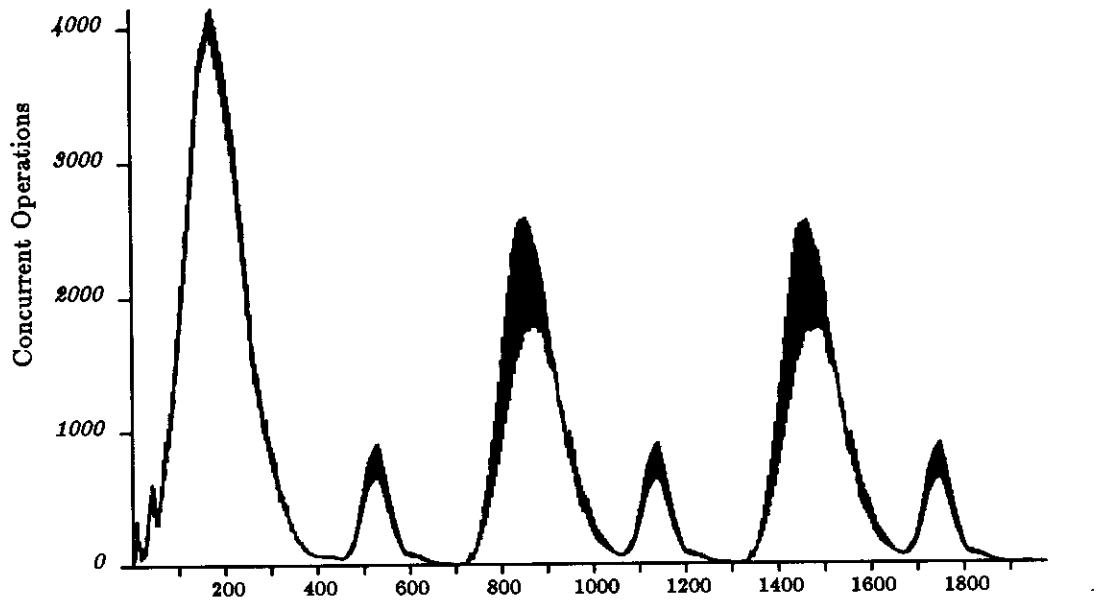


Figure 8: $(p = \infty, l = 0)$ Parallelism Profile for SIMPLE (3 iterations, $20 \times 20$).

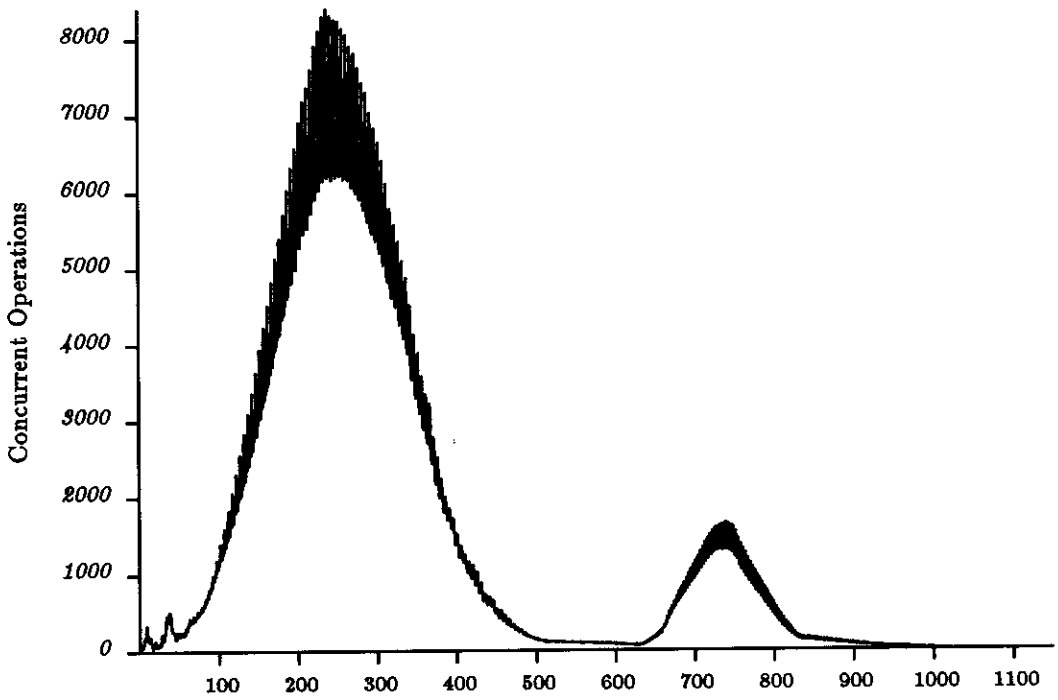


Figure 9: $(p = \infty, l = 0)$ Parallelism Profile for SIMPLE (1 iteration, $32 \times 32$).

algorithms employed and the compilation technology. Factors which place additional constraints in any realistic implementation, such as multiple simultaneous reads to an array component, communication latency, locality, contention, and distribution of work are abstracted away and do not affect the parallelism profile. Profiles shown thus far provide an upper bound on the parallelism. Though this bound is not achievable, it still provides insight into the application. For example, 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. But more importantly, we have come to believe that any large program that runs for a long time on current supercomputers has sufficient fine-grain parallelism to keep hundreds of processors utilized.

# 5 Parallelism Profiles on a 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 realistic machine, *i.e.,* a machine with finite number of processors and non-zero communication latency. We are unlikely to be interested in the maximum number of processors potentially active at one time, or the length of the critical path. Even average parallelism can be a misleading indicator of performance. In this section, we focus on the question of whether a given program has *sufficient parallelism* for an $p$ processor machine with latency $l$. A related question is what speedup is to be expected as more machine resources are provided.

We present two approaches to characterizing the behavior of programs on a finite number of processors. The first modifies the ideal machine, *i.e.,* the graph interpreter, 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.

## 5.1 The "Finite-Processor" Execution Model

As the first step towards a realistic execution model, we impose a limit of $p$ operations to be executed during each step. This represents $p$ processors only in a very abstract sense. We do not assign activities to processors, but rather choose up to $p$ 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 $p$ operations are enabled, but not all $p$ can be processed because some are on the same processor. In addition, we introduce communication latency by assuming that the output of every operator takes $l$ steps to be delivered to its destination.

In our abstract model, we assume that the communication system is highly pipelined, and has as much bandwidth as is needed. Thus, if an operator can produce a token at every step, the communication system can deliver them at the destination at every step after incurring an initial delay of $l$. Modeling I-structure operations requires care, since a `select` is a split-phase transaction; the token produced by the `select` operator is sent to the I-structure and, once the data is available, another token carrying the value is sent back to the operator waiting for the result. Thus, the result of a `select` is available at the destination operator $2l + 1$ units after the later of the `select` and the store for the particular element.

Modeling latency in this manner avoids the issues of network congestion and topology. For realistic architectures, the latency a token may experience is determined by the number of stages in the processor and storage-controller pipelines, as well as the number of stages in the communications network, which is expected to grow as the log of the number of processors. Thus, $l$ and $p$ are really

not independent parameters. However, we think it is better to estimate $l$ for a real machine by taking a weighted average of latencies experienced by tokens taking different paths.

In summary, the ($p$,$l$) or the *finite-processor non-zero latency* machine has the following characteristics:

1. All operations take unit time,
2. At most $p$ operations can be performed in a step,
3. Communication takes $l$ steps, and
4. Enabled instructions are selected through "fair" scheduling.

Ignoring latency for the moment, the finite processor restriction will not alter behavior of a program whose parallelism profile is less than $p$ throughout. When the parallelism exceeds $p$, the excess enabled instructions are thrown into a pool with the newly enabled instructions, and the instructions for the next step are chosen from this pool. For example, consider the dataflow graph and the parallelism profile in Figure 1. Suppose we set $p$ to 2. At step 3, we will only be able to execute two out of the three enabled instructions (*i.e.,* instructions 4, 5 and 6). Suppose we choose instructions 4 and 5. Then, the pool of instructions available for step 4 will contain instructions 6 and 7 (instruction 1 must execute after instruction 6). If, instead, we had chosen instructions 5 and 6 at step 3, then the pool for step 4 would contain instructions 1 and 4. In general, the choice of instructions from the pool at each step affects the finite-processor parallelism profile. However, the variation is negligible, provided the scheduling policy is *fair*, defined as follows.

Let $S_i$ and $S_{i+1}$ be the set of operators that execute in the $i^{th}$ and $i + 1^{th}$ steps, respectively, on the infinite-processor machine. A *fair schedule* for the finite-processor model is one in which all operators in $S_i$ are scheduled before any operators in $S_{i+1}$. In the profiles shown below, the particular fair schedule used by GITA, our graph interpreter, is determined by the way the compiler numbers destination arcs. It does not, for example, favor operators on the critical path.

The effect of execution on $p$ processors may be visualized by drawing a horizontal line at $p$ on the parallelism profile and "pushing" all the instructions which are above the line to the right and below the line. Figure 10 shows the profile for SIMPLE generated under this model with $p = 1000$, slightly greater than the average parallelism. The length of the critical path is increased from 1,976 to 2,763.

The effect of latency in the infinite processor model is uninteresting because the profiles are unchanged, except for gaps of $l$ steps between each of the steps in the zero-latency profile. The critical path length is simply dilated to $(1 + l) \times T_\infty$, where $T_\infty$ is the critical path length on the ideal machine.

When there are a finite number of processors, "excess" parallelism, *i.e.,* $pp(\tau) > p$, can be used to mask delays due to latency, assuming dataflow instruction scheduling. Again, consider the parallelism profile in Figure 1. Suppose we set both $p$ and $l$ to 1. In step 1, we execute instruction 1, after which we must wait one step before executing either instruction 2 or instruction 3. Suppose we execute instruction 3 at step 3, and instruction 2 at step 4 when the result of instruction 3 is in transit to its destinations, *i.e.,* instructions 4, 5 and 6. Thus, excess parallelism has been used to fill the gaps due to latency.

Figure 11 shows two parallelism profiles for 3 iterations of SIMPLE on a 20 x 20 mesh, with at most 100 operations per step. The top profile has latency $l = 0$ while the bottom has latency $l = 10$. Note that in this example, *the length of the critical path is increased only by a factor of 2 when latency is increased from 0 to 10.* These profiles suggest that the program has enough parallelism to absorb significant latency on 100 processors, if the processors are designed to take advantage of fine-grain parallelism.

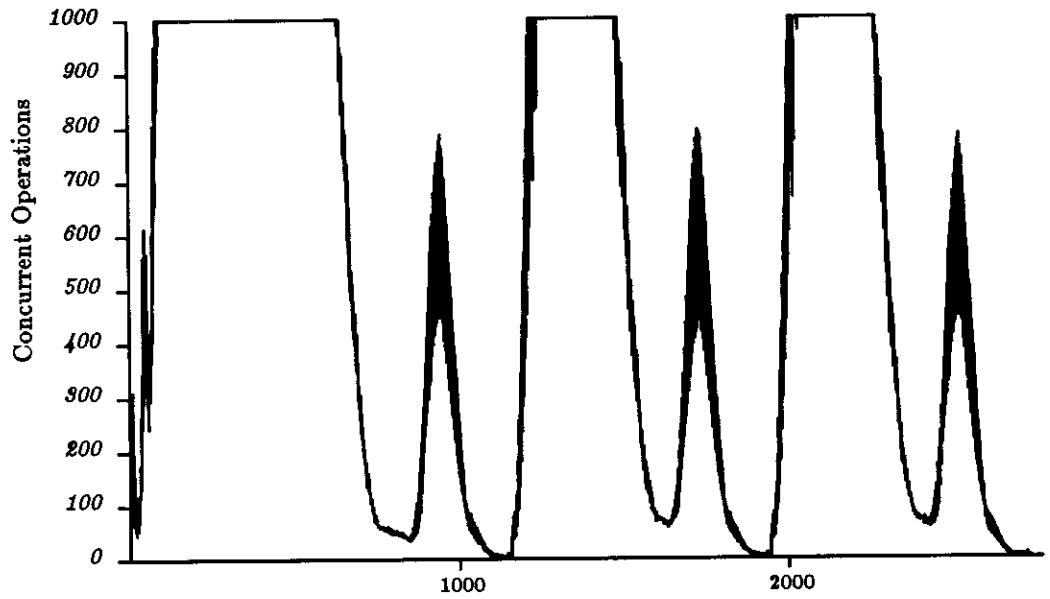


Figure 10: $(p = 1000, l = 0)$ Parallelism Profile for SIMPLE (3 iterations, $20 \times 20$).

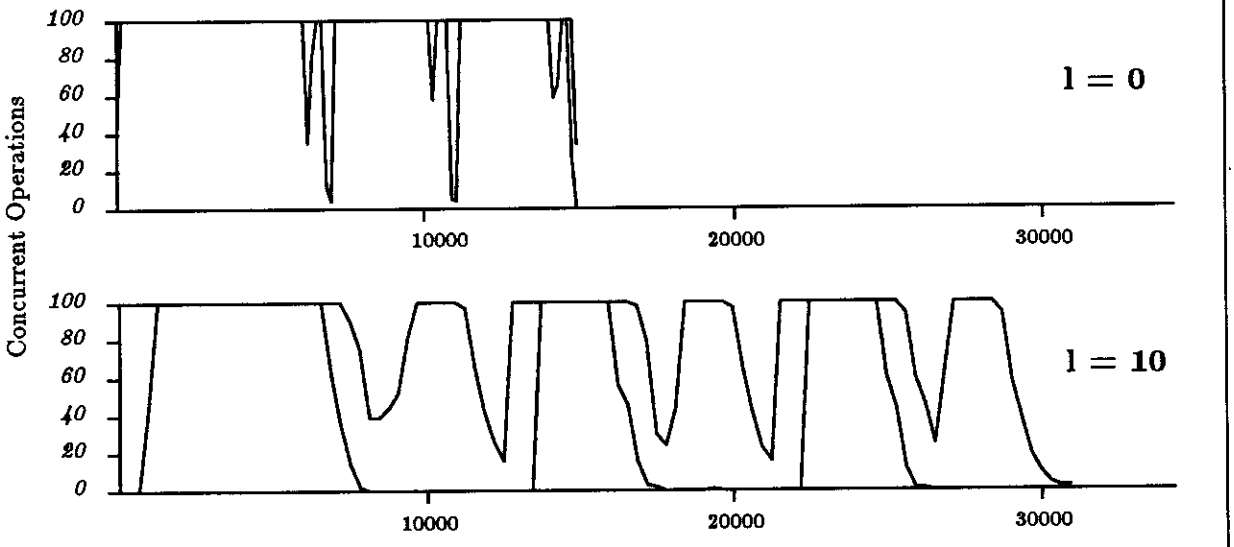


Figure 11: Effect of Latency on Parallelism Profile for Simple (100 processors, 3 iterations, $20 \times 20$).

## 5.2 Speedup and Utilization

The effective parallelism available in a program under the finite-processor model can be expressed in terms of *speedup* and *utilization*, as follows. Let $t(p,l)$ be the number of steps required to execute the program with at most $p$ operations per step and fixed communication latency of $l$ steps. Recall from Section 2 that $T_1$ is the total number of operations executed, *i.e.*, the area under the parallelism profile. Therefore, $T_1$ is simply $t(1,0)$.

$$speedup(p,l) = \frac{T_1}{t(p,l)} \qquad utilization(p,l) = \frac{T_1}{p \times t(p,l)}$$

These numbers tell us the limits on improved performance imposed by data dependencies in the algorithm itself, modulo influences of instruction scheduling. For example, for 3 iterations of SIMPLE ($20 \times 20$), $speedup(100,0) = 97$, and $utilization(100,0) = 97\%$. Thus, even on an idealized machine, *i.e.*, one with instantaneous communication and synchronization, we see that it is not possible to utilize all the processors all of the time. The utilization for this problem drops to 53% for $p = 1000$.

The explicitly plotted points in Figure 12 (we will discuss the curves momentarily) show the speedup measurements, at various settings of $p$ and $l$, for OLD-SIMPLE, which is the same program as SIMPLE, but coded in a previous version of Id. (Since procedure calls were strict in old Id, the potential parallelism in the program is reduced significantly. A direct comparison is difficult because the compiler for the earlier version used fewer optimizations).

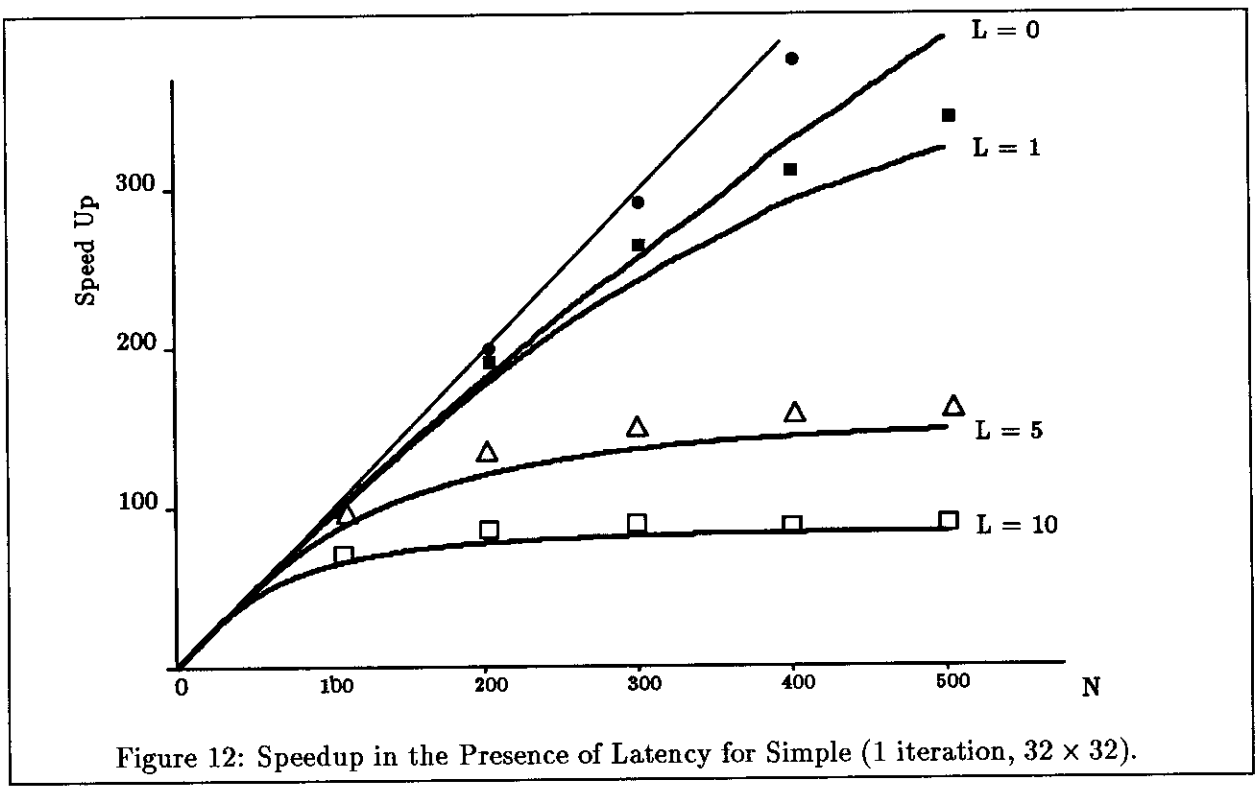


Figure 12: Speedup in the Presence of Latency for Simple (1 iteration, $32 \times 32$).

It is possible to derive asymptotes of the speedup curves, by observing that a machine with

latency $l$ cannot execute the program in less steps than $t(\infty, l)$, *i.e.*, $(1 + l) \times T_\infty$. Thus,

$$speedup(p, l) = \frac{T_1}{t(p, l)} \leq \frac{T_1}{(1 + l) \times T_\infty} = \frac{\overline{P}}{1 + l}$$

This shows the trade-off between average parallelism and latency, *i.e.*, we need more parallelism in a program to achieve the same speedup, if the latency is increased.

### 5.3 Estimating Speedup from the Ideal Parallelism Profile

Our interpreter, GITA, running on Lisp machines, took several hours to compute each point in Figure 12. Instead, an *estimate* of the speedup curve can be made from the ideal parallelism profile, $pp(\tau)$. Observe that for any time step $\tau$, if $pp(\tau) \leq p$, all $pp(\tau)$ operations can be performed in one step. However, if $pp(\tau) > p$, then we assume it will take the least integer greater than $\frac{pp(\tau)}{p}$ steps to perform $pp(\tau)$ operations. This is conservative, because it does not recognize that operations from the $\tau + 1^{st}$ step may be performed along with the last few operations of $pp(\tau)$.

We must consider the time at which tokens reach their destinations. A conservative estimate is to take $\left\lceil \frac{pp(\tau)}{p} \right\rceil + l$ as the time required to perform the computation in step $\tau$ and communicate its results. However, this is too conservative to model the behavior of dataflow execution, because we expect the tokens produced by the first $p$ operations to be in transit while the next $p$ operations of $pp(\tau)$ are performed. Thus, if $l \leq \left\lceil \frac{pp(\tau)}{p} \right\rceil$, we assume the computation of $pp(\tau+1)$ starts immediately after the last operations for $pp(\tau)$, since the earliest results from $pp(\tau)$ are likely to have arrived at their destinations by this time, with others following in pipelined fashion. Otherwise, we wait $l$ steps from the completion of the first $p$ operations of $pp(\tau)$ before computing $pp(\tau + 1)$. This estimate is optimistic; consider the pathological case where all the operations of $pp(\tau+1)$ depended on all operations of $pp(\tau)$. Under our optimistic assumptions about dependencies, we get

$$t(p, l) \approx \sum_{\tau=1}^{T_\infty} \max\left(1 + l, \left\lceil \frac{pp(\tau)}{p} \right\rceil\right). \tag{1}$$

Figure 12 shows the speedup curves derived in this manner. It can be seen that, in spite of our optimistic assumptions about data-dependencies, our estimate is conservative. We can get a somewhat more optimistic estimate by dropping the ceiling function.

## 6 Parallelism and Speedup With Program Partitioning

In characterizing and analyzing the inherent parallelism in programs, 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, simplify resource management, tag manipulation, and handling of loop invariants[2]. Advocates of hybrid von Neumann/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 [14, 16]. If the hardware is designed properly, then such "macro" operators can handle the "local state" more efficiently than pure dataflow.

Let us call a collection of operations which are constrained to execute on one processor a *task*. A task denotes a unit of work to be distributed. We allow an unbounded number of concurrent tasks, and tasks do not compete for processing resources. Within a task, instructions are *not statically ordered* by the compiler. Instead, they become enabled based on availability of data, and are scheduled fairly, as in the finite processor model. The 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 one operation is performed from each task that has any enabled operations. In this way, we isolate the effects of intra-task scheduling constraints without introducing additional effects due to the mapping of 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 von Neumann/Dataflow approaches. If the 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 unpredictable communication latency and synchronization waits.

## 6.1 Parallelism Under Different Task Granularities

Syntax suggests two natural places to partition programs: code-blocks, *i.e.*, instances of a loop or user-defined function, and, for a finer grain, iterations. Although we do not consider it here, one could imagine partitioning the program to minimize the number of arcs crossing the cut, ignoring the syntactic structure of the program.

We can characterize the inherent parallelism of the program under each task granularity through parallelism profiles generated on an unbounded-processor, *task-oriented* machine with the following characteristics:

1. All operators take unit time,
2a. At most one operation is performed per step per task,
2b. Operations from any number of distinct tasks may be performed in one step,
3. Communication is instantaneous, and
4. Fair scheduling within each task.

On such a machine, it makes sense to talk of *task-pp($\tau$)*, or task parallelism profiles. Figure 13 shows the ideal profiles for one iteration of OLD-SIMPLE $32 \times 32$ under the instruction-level finite-processor model and under the two task granularities [19]. 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 instruction to code-block granularity, the more important indicator, critical path, increases 8.4 and 32 times, respectively. Thus, the average parallelism is 1/8 and 1/32 of that under the ideal instruction-level model. These factors represent the loss in *potential parallelism* arising from the intra-task scheduling constraints. Within a task, the benefits of dynamic instruction scheduling are still realized, so these numbers represent the "best-case" using coarse-grain parallelism.

## 6.2 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 parallelism profiles show 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 profiles using Formula (1), taking $l = 0$. Figure 14 shows the estimated speedup from parallelism profiles of OLD-SIMPLE

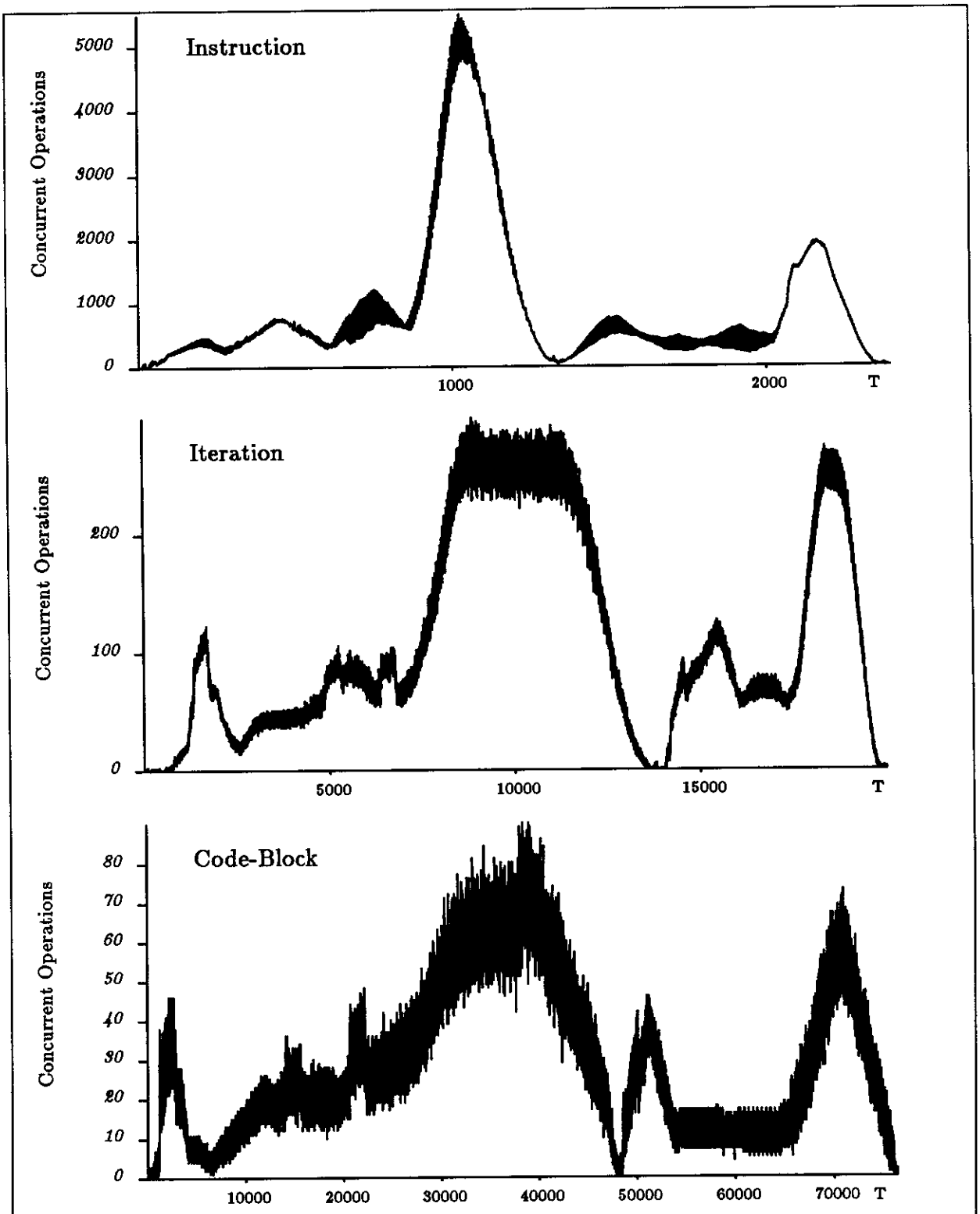


Figure 13: Ideal Parallelism Profiles of OLD-SIMPLE(32) under Various Task Granularities.

$32 \times 32$, $50 \times 50$, and $64 \times 64$. Note that for an $n \times n$ mesh the plateau in the curves of the code-block-level partition indicates that approximately $n$-fold parallelism is available.

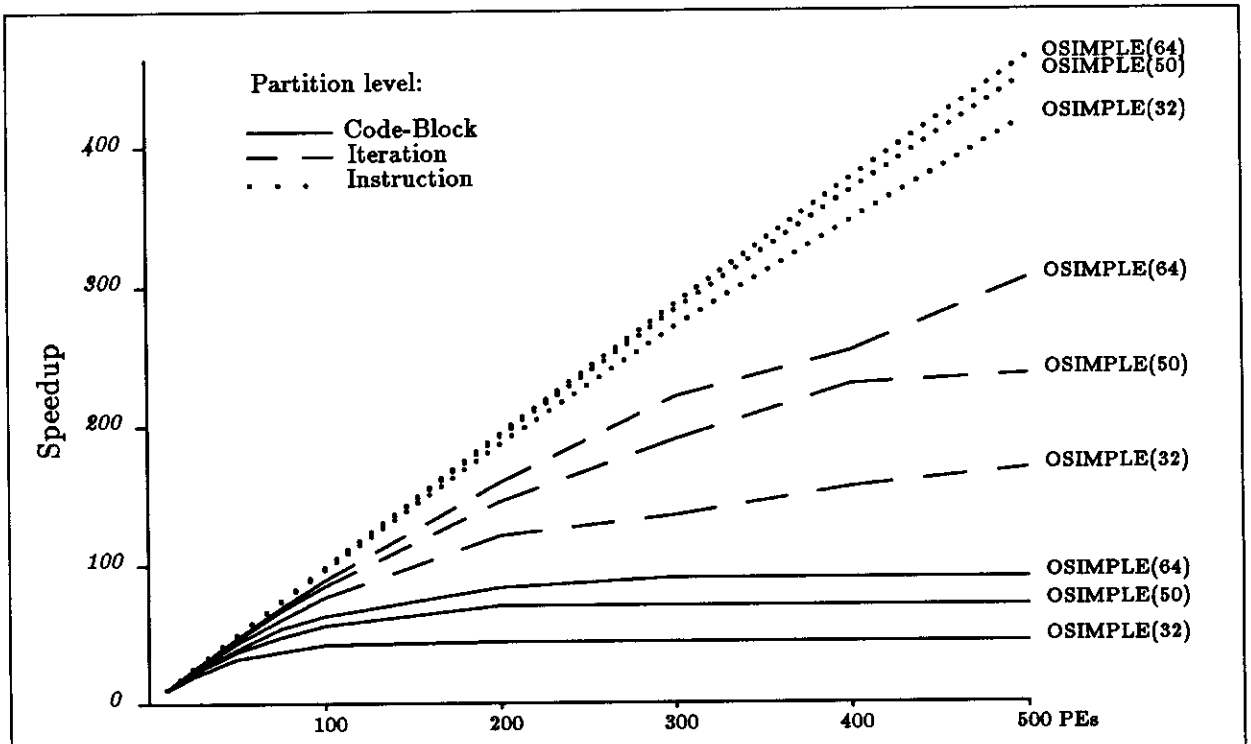


Figure 14: Speedup of OLD-SIMPLE(32-64) as Computed from Its Parallelism Profiles.

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* machine model with the following characteristics:

1. All operators take unit time,
2a. At most one operation is performed per step per task,
2b. Operations from no more than $p$ tasks may be performed in one step,
3a. Communication is instantaneous within a task,
3b. Communication requires $l$ steps otherwise,
4a. Fair scheduling within each task, and
4b. Round-robin scheduling amongst tasks with enabled operations.

Under this finite-processor task-oriented model, we observe the speedups shown in Figure 15 on OLD-SIMPLE $32 \times 32$ 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[1], 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, with $l = 10$ 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 processors and at a large number of processors given the problem

size.

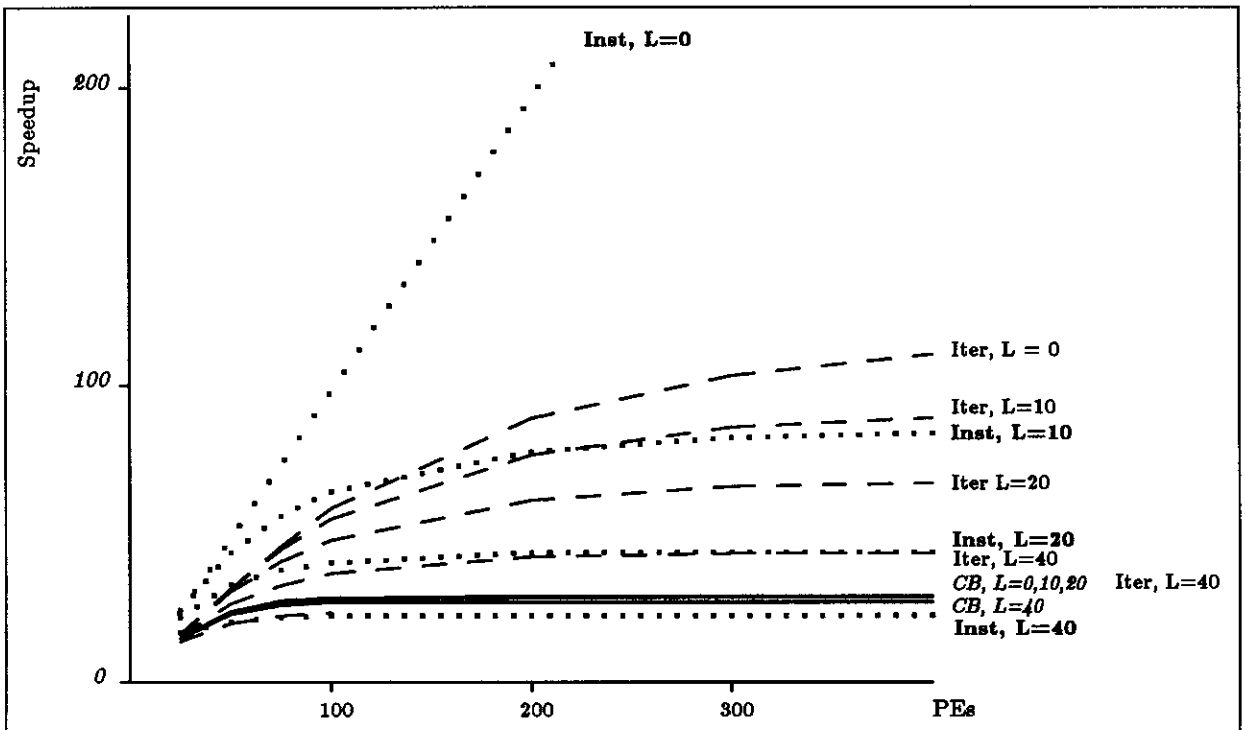


Figure 15: Speedup under various task granularities and latencies for OLD-SIMPLE 32 x 32.

## 6.3 Asymptotic Speedup of Programs under Different Partition 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 partition granularity in OLD-SIMPLE. The behavior of the program is nearly periodic, as there is little useful overlap between iterations.[3] 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 OLD-SIMPLE with various mesh sizes under the three granularities. Figure 13 represents the middle column. Looking across the columns we see how the *average parallelism* changes with problem size.

| Mode | 10 x 10 | 16 x 16 | 32 x 32 | 50 x 50 | 64 x 64 |
|---|---|---|---|---|---|
| Instruction $\overline{P}$ | 178 | 361 | 922 | 1584 | 2108 |
| Iteration $\overline{P}$ | 19 | 40 | 110 | 199 | 270 |
| Code-block $\overline{P}$ | 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.

[3] Other work [11] shows that there is, in fact, a hazardous kind of overlap between iterations, but that this can be removed.

| Ratio | 10 x 10 | 16 x 16 | 32 x 32 | 50 x 50 | 64 x 64 |
|---|---|---|---|---|---|
| Instruction $\overline{P}$ / Code-block $\overline{P}$ | 22.3 | 27.8 | 32.9 | 35.2 | 35.7 |
| Instruction $\overline{P}$ / Iteration $\overline{P}$ | 9.9 | 9.0 | 8.4 | 8.0 | 7.8 |
| Iteration $\overline{P}$ / Code-block $\overline{P}$ | 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.

# 7 Conclusion

We have presented a method for quantifying the parallelism in real programs developed in the context of a dataflow model, but the method should be useful in providing insights into other systems as well. 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 any specific implementation. We have presented two methods for deriving upper-bounds on potential speedup, and presented data on how speedup is affected by latency and intra-task scheduling constraints. This work brings us closer to answering questions of the form, "Does this program have sufficient parallelism for a machine of $p$ processors?", considering the characteristics of the program and the characteristics of the machine. Studies using the techniques presented here have substantiated our belief that large dataflow programs, even employing traditional algorithms, have "sufficient" fine-grain parallelism for machines of reasonable size. We have examined the speedup on a detailed simulation of the MIT Tagged-Token Dataflow Architecture. However, these experiments have necessarily been limited, due to the enormous space required for even small problem sizes. We run out of space on a stand-alone IBM 4381 on a 10 x 10 run of OLD-SIMPLE! To gain further insight, it is essential to conduct experiments on real dataflow machines, such as Sigma-1 [15] and Monsoon [22], which is now under construction at MIT.

Elsewhere [4] we have argued that efficient synchronization and dynamic scheduling of instructions are fundamental to general-purpose parallel architectures. Efficient synchronization is clearly essential to exploit producer-consumer parallelism. Here, we have shown the benefits of such an approach— by exploiting all forms of parallelism, ample parallelism is made available in most programs. Further, we have shown that a portion of this parallelism can be "invested" in masking communication latency, again relying on efficient low-level synchronization. Coarse-grain partitioning of programs may be advantageous to reduce communication requirements, but we claim that fine-grain parallelism is still required within a partition to mask latency. Also, we have tried to quantify how much parallelism is lost with such partitioning.

The goal of our research has been to show that dataflow architectures and languages can lead to general-purpose, programmable, parallel computers. This paper has presented only one aspect of our efforts to show the viability of dataflow. The programmability of these machines is demonstrated in [3]. Furthermore, in [1] we have shown that the cost of fine-grain parallelism, measured in terms of the number of extra instructions executed, is roughly a factor of 2 to 3. Development of

resource management strategies [10] and construction of what we believe will be a cost-effective, high-performance dataflow machine are currently underway [22].

**Acknowledgements:** Words cannot express our appreciation for Rishiyur S. Nikhil, without whom this paper could not have been completed. His thoughtful comments and advice, not to mention LaTeXpertise, have made this paper worth publishing.

## References


[1] Arvind, D. E. Culler, and K. Ekanadham. The Price of Fine-Grain Asynchronous Parallelism: An Analysis of Dataflow Methods. In *Proceedings of CONPAR 88, Manchester, U.K.*, September 12–16 1988. (to appear).

[2] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. 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.

[3] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987. (To appear in *Journal of Parallel and Distributed Computing*).

[4] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25-29 1987.

[5] Arvind and R. 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.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, February 1987. (Also to appear in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[7] Arvind, R. S. Nikhil, and K. K. Pingali. Id Nouveau Reference Manual, Part II: Semantics. Technical report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[8] A. E. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *COMPUTER*, 14(9):18–27, Sept. 1981.

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

[10] D. E. Culler. *Effective Dataflow Execution of Scientific Programs.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, Sept. 1988 (expected).

[11] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30 – June 2 1988.

[12] J. 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).

[13] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. Technical Report YALEU/DCS/RR-253, Yale University, Department of Computer Science, Apr. 1983.

[14] 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, Dec. 1987.

[15] K. Hiraki, S. Sekiguchi, and T. Shimada. System architecture of a dataflow supercomputer. Technical report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[16] R. 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, May 1988.

[17] H. F. Jordan. Performance Measurement on HEP - A Pipelined MIMD Computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden.* North-Holland Publishing Company, June 1983.

[18] L. Lamport. The Parallel Exection of DO Loops. *Communications of the Association for Computing Machinery*, 17(2):83–93, Feb. 1974.

[19] G. 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, Dec. 1987.

[20] R. S. Nikhil. Id Nouveau Reference Manual, Part I: Syntax. Technical report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[21] R. S. Nikhil. Id World Reference Manual. Technical report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[22] G. 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, July 1988 (expected).

[23] R. M. Russell. The CRAY-1 Computer System. *Communications of the Association for Computing Machinery*, 21(1):63–72, January 1978.

[24] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge, MA*, pages 202–211, August 4-6 1986.

[25] K. 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.

# Contents