Parallel Processing in Intermediate-Level Computer Vision

Pierre P. Tremblay

b. Eng. (Hons.), McGill University, 1989

Department of Electrical Engineering
McGill University
Montréal
July, 1992

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Engineering

Abstract

The problem investigated in this thesis is that of giving intermediate-level vision researchers adequate parallel processing tools for their work, where data and computational structures do not fit the SIMD execution model, but require a MIMD execution model instead. The contribution of this thesis is a comparison of 3 general-purpose MIMD parallel processing systems as tools for intermediate-level vision, by evaluating them against criteria which capture the essential issues in programming intermediate-level vision algorithms on such machines. According to my criteria, the best-suited of the 3 systems is composed of the Id functional language running on the Massachussetts Institute of Technology's Tagged-Token Dataflow Architecture (M.I.T. TTDA).

Scientific programmers will usually be refining the abstract models and algorithms which their programs implement, and should not be expected to be parallel architecture experts. Therefore, their tools should be adapted to their problem domain, for fast coding, and should provide logical independence from solving the four crucial MIMD issues of parallel processing, partitioning, scheduling, synchronization, and memory latency. Current tools, such as Uniform System programming on the BBN Butterfly and C-Linda programming on the Sequent Balance, do not

Functional languages, on the other had, as exemplified by the Id language running on the TTDA architecture, are a more appropriate solution. They are close to the scientific problem domain, because they are based on the function and on expressions. They are amenable to compiler solutions of the partitioning, scheduling, synchronization and memory latency problems, because they do not allow a programmer to specify restrictive commands, or to artificially restrict the order of execution. This allows a compiler to extract all the parallelism present in a program, which is necessary to obtain good performance on highly parallel machines, such as the TTDA

Résumé

Le problème abordé dans ce mémoire est celui de fournir aux chercheurs en vision numérique de niveau intermédiaire les outils de traitement parallèle nécessaires pour leur recherche. Leurs algorithmes comportent souvent des structures de données et de calcul qui se prêtent mal au modèle d'éxécution *SIMD* (instruction unique, données multiples) et beaucoup plus au modèle *MIMD* (instruction multiples, données multiples) La contribution de ce mémoire est la comparaison de trois systèmes de traitement parallele *MIMD* en tant qu'outils pour la vision numérique de niveau intermédiaire, en les évaluant selon des critères qui font ressortir l'essentiel des éléments requis dans la programmation d'algorithmes de vision de niveau intermédiaire sur ces machines. Selon ces critères, le systeme le plus approprié est composé du langage fonctionnel Id éxécutant sur l'architecture TTDA du M.I.T.

Les programmeurs-chercheurs sont habituellement intéressés a améliorer leurs modèles et algorithmes, et non pas à devenir experts en archtectures parallèles. Conséquemment, leurs outils devraient être adaptés à leur domaine de recherche, pour faciliter le codage, et devraient créer une démarcation logique qui sépare le programmeur des quatre considérations cruciales propres aux systèmes MIMD, le découpage, l'ordonnancement, la synchronisation et le temps de latence de la mémoire. Les outils disponibles présentement, tels que la programmation avec le *Uniform System* sur le *BBN Butterfly* et la programmation en C-Linda sur le *Sequent Balance*, ne le font pas.

Par contre, les langages fonctionnels, comme le langage Id éxécutant sur l'architecture TTDA, représentent une solution plus appropriée. Ils sont plus près du domaine scientifique, étant basés sur la fonction et sur les expressions mathématiques. Ils se prêtent beaucoup mieux à des solutions par compilateur des quatres problèmes mentionnés plus tôt, parce qu'ils ne permettent pas au programmeur de spécifier soit des commandes ou un ordre d'éxécution restrictifs. Ceci permet à un compilateur d'extraire tout le parallélisme présent dans un programme, ce qui est nécessaire pour obtenir une bonne performance sur des machines hautement parallèles, telles que le TTDA.

Acknowledgements

Even though a single name appears on the front page of a Master's thesis, that person knows very well that he/she could not have made it without a great deal of help, academically, technically, morally, and of course financially

I must first thank my thesis supervisor, Frank Ferne, whose enthusiasm and openness I appreciated throughout. I also have to thank him for his excellent suggestions that restructured this thesis into a far clearer paper than it originally was.

I am also deeply indebted to Dr Guang Gao, who stimulated my interest in parallel processing through courses and discussions He also provided access to two of the three parallel processing systems used in this thesis, the BBN Butterfly and Id World. Without these, this thesis would simply not have been possible, and I offer my sincere thanks

Thanks are also due to Josef Fritscher, of the Computer Center of the Technical University of Vienna, Austria. He arranged for me to use C-Linda on their Sequent Balance multiprocessor through the global Internet It is one thing to know that it is technologically common-place to have a program run across the Atlantic ocean and display results in North America, but quite another to experience it!

Locally, I must thank Lee Iverson, Peter Whaite and Andre Foisy, for technical support and advice, especially during my early days at McRCIM. The extent to which they contributed to my knowledge cannot be understated

Je dois aussi remercier le Fonds FCAR pour leur appui financier au cours de mes études.

Enfin, je voudrais dire merci à ceux qui me sont chers, Marie-Claude, ma mère, mon père, ma soeur, mon frère, et le reste de ma famille. Votre amour et votre encouragement m'ont toujours été précieux.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Among the challenges of computer vision is the tremendous amount of processing that must be done to extract desired information. Current computer vision research is often hampered by the long turnaround in experiments caused by large processing requirements. This has quite naturally motivated vision researchers to explore parallel processing as a tool for making certain problems in computer vision research tractable. The contribution of this thesis is a comparison of 3 general-purpose parallel processing systems as tools for intermediate-level vision. I will show that the best-suited of these is composed of the Id functional language running on the Massachussetts Institute of Technology's Tagged-Token Dataflow Architecture (M.I.T. TTDA)

Scientific programmers in general, and computer vision programmers in particular, will usually be refining the abstract models and algorithms which their programs implement, and should not be expected to be parallel architecture experts. Therefore, their tools should be adapted to their problem domain, for fast coding, and should provide logical independence from solving the four crucial MIMD issues of parallel processing, partitioning, scheduling, synchronization, and memory latency [19]. Current tools, such as Uniform System programming on the Butterfly and C-Linda programming on the Balance (the other two systems we examine), do not, in spite of the fact each is available commercially and in general use as a general-purpose MIMD scientific problem-solving tool.

Functional languages, on the other had, as exemplified by the Id language running on the TTDA architecture, are a more appropriate solution. They are close to the scientific problem domain, because they are based on the function and on expressions. They are amenable to compiler solutions of the partitioning, scheduling, synchronization and memory latency problems, because they do not

allow a programmer to specify restrictive commands, or to artificially restrict the order of execution This allows a compiler to extract all the parallelism present in a program.

I will first discuss in this chapter what the current problems are with intermediate-level vision parallel programming, then propose criteria by which I feel parallel processing systems used for intermediate-level vision should be judged. I then propose functional languages as a solution which matches the criteria. In chapter 2, I describe the 3 systems I have chosen to examine. Uniform System programming on the BBN Butterfly, C-Linda programming on the Sequent Balance, and Id programming on the M.I.T. TTDA. In chapter 3, I construct a plausible intermediate-level vision problem, implement it on all 3 systems, then discuss how each performed under my evaluation. In chapter 4, I show some of the lessons that should be drawn from this exercise Concluding remarks are given in chapter 5.

1.1. What is the Problem?

Computer vision research, and the development of new algorithms and techniques for image analysis, is typically an iterative process, where an algorithm is proposed to implement an abstract model, and, in turn, the algorithm is implemented with a computer program. In basic research, the interest is mainly in refining the abstract model or the algorithm, and less often the program itself. Thus, fast program execution (performance) is needed to improve experiment turnaround, but programmability is also essential, as code is likely to evolve rapidly, in some instances being replaced entirely because of changes in the abstract model. Code development is less likely to be amortized over long program life spans, and must therefore be relatively inexpensive.

Tools for handling the one-to-one mapping of input pixel to output data in low-level vision are well understood. Partitioning, scheduling and synchronization of tasks on processing elements is straightforwardly and effectively performed on SIMD (Single Instruction Multiple Data) machines. Parallelizing other levels of computer vision is less obvious. We will look at the many-to-one (or iconic to aggregate) mappings typical of many vision algorithms, which we will collectively refer to as intermediate-level vision. In such cases, the non-uniform distribution of output features and

obviously data- ("feature"-) dependent nature of processing renders the SIMD approach ineffective. Instead, we require more general partitioning, scheduling, and synchronization, which are found in the MIMD (Multiple Instruction Multiple Data) processing model. None of these issues are simple ones for the scientific programmer to deal with, as we shall see next.

1.1.1. Four Parallel Computing Issues

The four issues of greatest importance in the parallel execution of a program are [19, p. 26] *partitioning, scheduling, synchronization,* and *memory latency*. These are essential, because they are concerned with both the performance of the program and the scientific user's view of the computing process, instead of the engineering view of the hardware machine. *Partitioning* means specifying the sequential units of computation in a program, to find the partition size which strikes a balance between low overhead and high parallelism, and thus minimizes run-time. *Scheduling* is assigning tasks to processors to minimize run-time by optimizing processor utilization and inter-processor communication. *Synchronization* is a mechanism for coordinating the activity of processes; tasks working together must synchronize to coordinate producer-consumer relationships, forks and joins, and mutual exclusion. *Memory latency* is defined as the time between a memory request and the answer to that request. I discuss the problems arising from each issue in turn.

The partitioning problem can be understood in the following way. For high performance, we want *high parallelism* with *low overhead*. However, *increasing parallelism* brings *increased penalty* for synchronizing and scheduling additional tasks, and on the other hand, *reducing overhead* is done by *merging or fusing tasks,* which decreases the penalty for synchronizing and scheduling tasks, at the cost of *wasted parallelism*. The consequences of the partitioning/parallelism tradeoff [35, p. 15] are that the presence of overhead can make it impossible to achieve ideal speedup, and the real parallel execution time is minimized at an optimal intermediate granularity. The *partitioning problem* is to *find the corresponding optimal intermediate partition*.

Scheduling also involves tradeoffs [35, pp. 15–16], between parallelism and overhead. Parallelism dictates that tasks should be assigned to different processors as much as possible, but communication overhead is reduced when tasks are assigned to the same processor.

Synchronization will be detrimental to efficiency because of improper granularity [35, p. 7], if the synchronization granularity in the program is too fine for the target multiprocessor[1]

Memory latency becomes a big problem if the multiprocessor system is built out of von Neumann processors, which must either wait for the memory response, or do an expensive context switch. The memory latency problem often appears under the guise of the "data partitioning" problem, as data must be "partitioned" and placed on different memories so as to minimize memory latency.

These are the crucial issues that must be solved on a MIMD system for a parallel program to achieve good performance However, I strongly believe that placing the responsibility for solving these issues into the hands of the scientific programmer is counter-productive. The scientific programmer should not be expected to be a parallel architecture expert, and instead should be given a tool to do research. Freedom from dealing with the underlying architecture because of these issues is called *logical independence,* and this is what we seek to give to the computer vision programmer. Current parallel processing solutions rarely provide this independence, as we shall see later on.

1.1.2. A Definition of Intermediate-level vision

We can define intermediate-level vision as the category in which the input is a set of values still associated with each pixel (*i.e.* when an $n \times n$ pixel image has produced an $n \times n$ array of values, or pixel labels) and the output is a structure that is *not* a two-dimensional array (*e.g.* a list of features [37][2]).

A more general definition, perhaps better suited to the large variety of possible algorithms and tasks, is to view intermediate-level vision as a many-to-one mapping, or an iconic-to-aggregates transformation [17]. Intermediate-level vision is that part of the vision process that performs a reduction in the amount of information handled, abstracting out desired features. This phase of processing is present in all so-called image understanding systems.

[1] As of this writing, most multiprocessors cannot efficiently support more than 1 synchronization in 100 instructions (per processor).

[2] This definition must be slightly extended, of course, for truly three-dimensional images, such as those obtained from computed tomography

Intermediate-level vision can also be described in terms of its data and control structures [22, p. 743]. *Data structures* at input and output go from images to features (a *many-to-one mapping*). *Computational structures* during processin can include some or all of the following:

- numeric processing
- non near-neighbor communication (regional processes, a priori non diameter limited)
- potentially irregular communications (in destination and volume) *e.g.* feature extraction, image segmentation

Examples of intermediate-level processes include those for chain encoding, Hough transforms, shape measurement and description (such as convex hulls and others), building region-adjacency graphs [37], or aggregation (partitioning or linking) and model fitting [18].

Thus, a complete characterization of vision algorithms can be seen in figure 1.1 (adapted from [22]). Of course, this rigid categorization scheme should not be implied to fit all algorithms exactly; nevertheless, it is a useful tool to assess the needs of programmers implementing algorithms that should for the most part fit into it.

1.1.3. The Parallelization Characteristics of Intermediate-Level Vision

In this section we look at how the different levels of computer vision can be parallelized. One of the most important points of this thesis is the following: an intuitive, explicit intermediate-level vision partitioning scheme [37, p. 9] is not as straightforward as for low-level vision tasks, where image partitioning is the natural choice. Let us examine the reasons why.

We begin by looking at low-level vision algorithms, with respect to explicit partitioning (granularity), processing type, and algorithm type [22, p. 745] . *Partitioning* is typically image partitioning (*i.e.* one processing element for a regular group of pixels), usually at a fine granularity (*e.g.* one processing element per pixel). *Processing* proceeds in a data parallel and synchronous fashion: the same

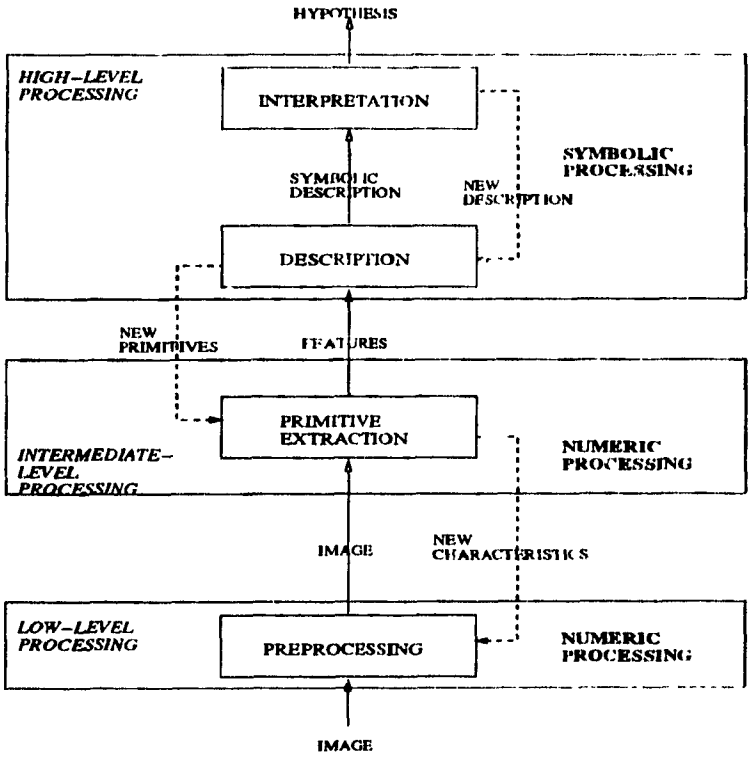


Figure 1.1: The different levels of computer vision processing, their inputs/outputs and characteristics

instruction is executed on parallel data, in lock-step across all processing elements. The *algorithms* employed are typically deterministic, and entirely numeric.

Current low-level vision architecture characteristics typically include [37, p 3] one processing element per pixel, and processing elements connected in a 2D mesh, executing synchronously in SIMD fashion Partitioning is straightforward, each task being assigned processing for a fixed region of the image, there is no data-dependent task creation, so a fixed partitioning is possible. Scheduling is also straightforward, as the interconnection of processors in the 2D mesh usually correspond quite well to the structure of communication in the algorithm, which is near-neighbor. Synchronization is implicit in the architecture model all processors execute synchronously, in lock-step. Memory latency is solved because each processor deals with data either in its local memory, or communicates with processors nearby to obtain the data it needs. Global, arbitrary communication patterns do not fit this execution model very well, but are not needed in low-level vision algorithms.

Intermediate-level vision [37, p. 9] is usually computationally intensive, as algorithms have to examine large numbers of input pixels, or data in one-to-one correspondence with pixels. The algorithms can also be looked at in terms of parallel characteristics such as partitioning (granularity), processing, and algorithm type [22, p 745–746] *Partitioning* is in terms of image and function partitioning; either a regular group of pixels per processing element (image partitioning), or a (set of) function(s) per processing element (function partitioning). *Processing* proceeds in a control parallel (multiprocessing) fashion (each processing element decides of its own change of state and instruction execution), and is asynchronous (from one processing element to the other; explicit synchronization instructions are needed) *Algorithms* are primarily numeric

For some intermediate-level vision algorithms, since the input data consists of pixel labels assigned to each point, we can use image parallelism and partition the pixel label data equally among the processors This is the same scheme as in low-level vision, where output results are spatially distributed in the same manner as the input, and the amount of processing over an image area will not differ significantly from one area to the next. In intermediate-level vision however, objects being computed as output are distributed over the image, but seldom in a uniform way. Therefore, a simple partitioning and scheduling scheme where each task gets an identical-sized piece of the input image and is scheduled on a single processor will lead to load imbalances in a multiprocessor [37, p. 9], as features (the result of intermediate-level processing) in image are not uniformly distributed,

and vary in size and in time required to compute them; note especially how inappropriate the SIMD execution model would be It might also be the case that tasks are created dynamically (at run time), as a result of processing. Additionally, there is usually significant parallelism *within the algorithms* used to compute these features. This requires even more flexibility in dealing with the partitioning and scheduling problems

Explicit synchronization [37, p. 9] therefore becomes an important issue If spatial partitioning (spatial parallelism) is used, curves or regions may extend across image to fall into zones of several tasks, which must then coordinate and synchronize, to work on the same object simultaneously.

All these concerns arise because of the fact that partitioning (which includes process creation), scheduling (mapping) and data placement to avoid memory latency are, in intermediate-level vision, *data-dependent* and many-to-one, not fixed and one-to-one, as in low-level vision This will make it difficult for a programmer using an explicitly parallel programming language, who must match these requirements to the underlying architecture, to obtain good performance: it is difficult to know in advance whether good performance will be obtained by program execution on a particular data set.

Let me then summarize the issues in parallelizing intermediate-level vision algorithms Affecting partitioning and scheduling are spatially non-uniform feature distribution, variations in feature computation time, possible dynamic (data-dependent) task creation, and the need to exploit finer-grained parallelism within the feature computation. Synchronization must handle the fact that tasks must cooperate to compute features, and to handle the interactions between features. Memory latency considerations are that a large amount of iconic input data must be distributed to a large memory and shared by multiple processes.

Faced with these considerations, scientific programmers in general, and intermediate-level vision programmers in particular, must consider available options that will allow ease of coding and yet obtain good performance. These options are in terms of MIMD parallel hardware architectures and programming languages.

1.2. Shortcomings of Current Tools

Most parallel programming systems proposed to date for scientific needs use explicit parallelism (*i.e.* require the programmer to specify partitioning, scheduling and synchronization). There are three mechanisms which have been used to include parallelism in explicitly parallel programs [32]:

concurrent languages: incorporate parallel features as integral parts of a language's design.

language extensions: adding parallel extensions to an existing sequential language.

parallel runtime libraries: providing high-level interfaces to parallel routines stored in a system library

The scientific problem solving process for parallel programming is typically composed of *multiple restructurings* [32, p. 16]. from abstract model, to algorithmic solution, (manually) to program code, (automatically by compiler) to executable code. Each restructuring complicates program development, as it is a source of potential error and distortion, in addition to imposing development overhead. The first and third transformations pose no special problems to the scientific programmer: the *conceptual to algorithmic* transformation is most comfortable and best understood by the scientific programmer, while the *implementation to physical* transformation is performed by compiler technology The second transformation *(algorithm to implementation)* is the one that poses special difficulties for the scientific programmer In this transformation, the most critical factor is the programming language, which provides the framework for describing how the problem's solution will be achieved [6, p. 474]. A language design has the most significant impact on how easily an algorithm can be transformed into workable code. These transformations are shown in figure 1.2.

Scientific programmers currently rely on [32, p 18] language extensions and run time libraries, instead of learning new parallel languages. The reasons are that extensions to familiar sequential languages — through parallel constructs or high-level interfaces to libraries — are more likely to appeal to scientific programmers than are new concurrent languages [21, pp. 356–357], [32, p. 18]. There is also an apparent ease in parallelizing sequential programs using extensions or libraries, as there is minimal rewriting. A further reason is the availability of production-level compilers for parallel machines.

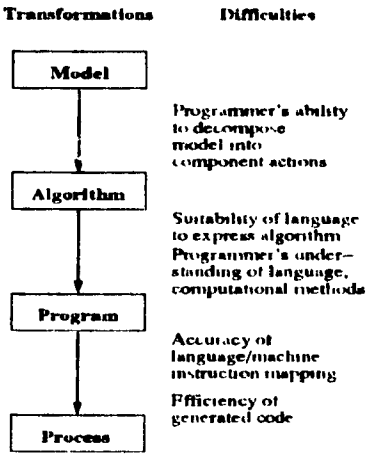


Figure 1.2: Scientific programming transformations From [32, p 17]

We first examine parallel libraries, or system libraries with routines for parallel execution. Why bother with parallel languages when operating system services or libraries can be provided to allow concurrent processes to coordinate? Because they are too crude a method for expressing parallelism [21, p. 355]. They show a *syntactic crudeness*, with messy, poorly integrated syntax, often with long parameter lists, because they exist in isolation from a user's program, and have no compile-time checking or optimizations. They also show *semantic crudeness:* it is harder for the programmer to provide operations that perform complex or sophisticated functions efficiently, because of the fixed nature of the mechanisms provided

Other explicitly parallel approaches, such as new parallel languages and extensions to sequential languages (also called *annotations*) suffer from problems as well. Message passing programming has been criticized as difficult and non-intuitive [21, p. 354]. For parallelization of scientific codes using both sequential language extensions [32, pp. 18–20] and libraries, the difficulties are typically the same. Parallelization of loops is the most common form of parallelization, and it is the programmer's responsibility to find solutions to data dependencies by partitioning data among iterations, a source of problems for programmers. The programmer is also responsible for explicit scheduling and synchronization, when global variables are altered by a task, which is tedious and error-prone In fact, the wrong annotations (for synchronization or communication) will make program non-deterministic (time- and machine configuration-dependent), a nightmare for debugging. Many scoping and storage management notions, such as local and remote memories, are counter-intuitive

to scientific programmers. The resulting code is often machine-specific and obscure [35, p. 1]. Restrictions on parallel constructs may require the reformulation of sequential control structures as well, producing more restructurings, more error possibilities, and less readability. In addition, the extended source languages (*e.g* Fortran, C) often suffer from lack of expressivity to begin with [6, p. 461]

The current trend appears to be towards languages with explicit partitions [35, p 11], or, in other words, languages based on the explicit tasks model, in both new languages and those based on sequential languages Compared to their predecessors, these languages offer portability but still force the programmer to explicitly decompose the program into tasks and control their synchronization and communication Languages with explicit partitions are easier to implement only at a large cost to the programmer. The programmer now has to worry about [35, p 11]

correctness: the programmer must avoid deadlock and race conditions (*i.e* ensure determinacy), which occur because of errors in inter-task synchronization and communication. These do not arise in sequential programs Errors based on race conditions are notoriously difficult to debug, or even reproduce [35, p. 11].

performance: the problem with explicit partitions in task-based languages is that the performance of a given partitioned program may vary dramatically over different multiprocessors, thus rendering the program non-portable in practice.

In terms of correctness, unstructured tasks are analogous to the GOTO's of sequential programming. In terms of performance, granularity considerations can clutter up the code and become an extra burden to the programmer. Compiler partitioning [35, p. 12] ensures portability to future multiprocessors, and a uniform programming model for programmers.

The consequences of choosing sequential language extensions and parallel libraries [32, p. 18] currently produce compromised program structural integrity: parallelism is added after the fact, in *ad hoc* fashion, adding yet another restructuring to program development. Additionally, the use of vendor-specific programming libraries or extensions means machine-dependent programs in their final form.

This is partly because of the deficiencies current parallel programming systems support. Programmers are forced to juggle potentially dangerous operations, as in many cases compilers can find and report only the most blatant errors. The primitives used to specify parallelism are usually closely tied to underlying machine. The management of architectural configuration is the full responsibility of the programmer, now concerned with solving *in her program* the four issues of parallel execution we have been mentioning, optimal partitioning, scheduling, explicit synchronization, and the distribution of data to memory locations to solve the memory latency problem. All of these determine the efficiency, effectiveness and reliability of the parallel implementation. In short, parallel systems lack the buffering effect of *logical independence*: parallelism should be incorporated at a reasonable level of abstraction, *rather than simply providing a notationally convenient way of specifying what are in fact machine-specific operations*. Instead, "parallel languages reflect a low-level view of concurrent execution that reinforces user misconceptions which increases expense of program development, and raises questions about reliability of parallel programs. In fact, the present level of language support for parallel programming requires that the user expend *more effort in managing the problem-solving resource than in actually solving the problem*" [32, p. 21].

Some of the problems arise from misconceptions by scientific programmers, who often ignore the effects of *nondeterminism* [32, p. 20]. The fact that a parallel program functions correctly once, or even one hundred times, with some particular set of inputs, is no guarantee that it will not fail tomorrow with the same inputs.

Faced with these problems, what can be said about the needs of scientific parallel programmers? Certain lessons can be drawn from sequential programming [32, p. 17]. First, move the algorithmic solution closer to the implementation, shift transformation responsibilities away from the programmer to the compiler and the parallel architecture, and finally provide *logical independence*, a clear delineation between the two levels of transformation (algorithm to program and program to physical), which represents a commitment to maintaining a separation between machine-dependent and machine-independent factors.

The lessons of three decades of sequential program development are clear: programmer effectiveness improves when language structures are moved away from physical issues and toward logical models. While computing professionals should be able to apply configuration-specific expertise, it is counter-productive to expect the same of the general user community. Scientific programmers

cannot be expected to solve issues of partitioning, scheduling, synchronization and memory latency every time they write a program to implement an algorithm, or move to a new architecture; it is simply too tedious. For reliability, determinate high-level constructs are needed, and we must move away from the inadequate expression of scientific applications in terms of a particular machine or memory model. Even if scientific programmers do code a determinate program, there is no guarantee their programming expertise and knowledge of a given system will produce an efficient implementation. Sequential programming has evolved to shield programmers from physical details and maintain logical independence, which has become a great strength; parallel programming would be well-advised to do the same [32, p. 23], [2]. In short, parallel computers must be made accessible to do science. Researchers must be free to concentrate on their research, not struggle with machine-dependent quirks and minute details [32, p. 23].

1.2.1. Implications for Intermediate-Level Vision Parallel Programmers

What do the intermediate-level vision characteristics given in section 1.1 mean for a scientific programmer who wants to code her algorithm on a parallel machine? What do the above general comments of sections 1.1 and 1.2 mean in the context of intermediate-level vision? We develop a partial framework to answer these questions in this section.

We first examine what the characteristics of intermediate-level vision algorithms imply in terms of requirements for parallel processing, with respect to parallel programming system support. How do the characteristics of intermediate-level vision algorithms affect the type of parallel programming language which should be used in implementing these algorithms? Specifically, we outline a number of criteria to evaluate parallel processing systems' appropriateness for intermediate-level vision parallel programming.

There are two initial requirements that are universally agreed upon:

1. **ease of programming:** for experimental algorithm design. Includes general applicability but with closeness to problem domain, determinacy of results, logical independence, *etc.*

2. **performance:** execution speed on a given architecture.

In our case, as we favor algorithm experimentation and quick prototyping of algorithms, programmability will be the most important of the two

The following characteristics in a programming system will partly satisfy these requirements. Each contributes to performance or programmability to varying degrees. A comparison metric that includes all of these criteria is difficult to create, because the relative weighing of the criteria is not easy. It is obvious from the list below that there is considerable overlap in some of the criteria for example, load balancing is a function of partitioning and scheduling, and flexible handling of large data structures is a function of whether or not it is the programmer's responsibility to solve the memory latency problem. Additionally, the first two criteria are somewhat orthogonal to parallelism; however, we feel they are important, as some parallel processing systems are not as well suited for the task as others. Figure 1.3 shows the relationships between these criteria and the four issues we have identified as crucial to parallel processing. The criteria are:

closeness to problem domain: for fast prototyping; in our case, we have numerical processing and mathematical algorithms, expressed in the mathematical notation of functions.

general applicability: within the intermediate-level vision problem domain, reasonable programmability for all problems, irrespective of communication patterns, task creation requirements, *etc.*

ease and flexibility in task creation: because of data-driven control flow in intermediate-level vision algorithms, task creation should be easy to code.

lightweight task creation: because of the massive parallelism present in intermediate-level vision algorithms, we don't want the creation of a task to be an expensive operation.

determinacy of results: invaluable for debugging.

easy load balancing: necessary because of the spatially non-uniform distribution of output features in the input data.

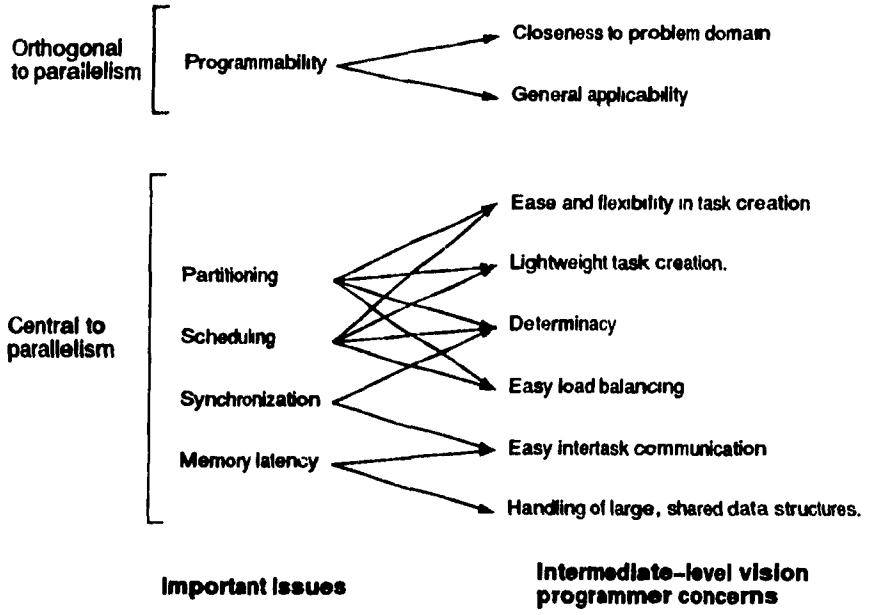


Figure 1.3: This figure shows the criteria for intermediate-level vision programming we have laid out, on the right, and their relationship to the more general issues of programmability, partitioning, scheduling, synchronization, and memory latency.

easy intertask communication: because communications can be irregular in destination and in volume. In particular, communication requirements will differ from low-level vision, in that we will not necessarily have near-neighbor communication.

handling of large, shared data structures: the system must permit flexible and transparent access to large, shared data structures (*e.g.* the input image).

In fact, many of these criteria are simply different ways of stating the need for logical independence, to simplify program development and avoid explicit partitioning, scheduling, synchronization, and memory latency concerns, and their problems. In the sections below, we examine how three given systems fulfill these criteria, through the programming of a sample algorithm.

1.3. Functional Languages: A Better Solution?

I feel that languages of the future will be successful because they will feature implicit partitions and schedules, and will not have the problems of explicit partitioning, or the problems of automatic parallelisation of today's imperative parallel languages. The main obstacle to widespread use of such languages on current multiprocessors is the problem of compiler partitioning and scheduling; see [35, p. 12] for a functional language solution

Functional languages offer a different path to parallel execution. If the goal of parallel programming is to have real-world problems mapped to parallel hardware seamlessly and automatically [32, p. 20], then two options are possible, parallelism detection in sequential imperative languages, or parallelism detection in declarative languages. However, for reasons mentioned in appendices B, C, and D, parallelism detection in sequential imperative languages, while very desirable from user standpoint, will fail to detect most of the parallelism present [6, p. 461] in a program.

Functional languages offer relief for both [6, p. 460] *high-level encoding* and *generation of efficient code* for the following reasons. Higher-order functions raise the level of programming, as well as encouraging the use of small functions that directly relate to the mathematical and physical concepts

of the problem. In the second case, the straightforward operational semantics of functional languages provide tremendous opportunities for parallel execution.

More specifically, functional language advantages are that their [6, p. 460]

- declarative nature eliminates overspecification of order of evaluation,
- their operational semantics automatically expose parallelism present in a program,
- their higher-order functions elevate level of programming so that abstractions can be built closer to the concepts in the problem domain,
- they produce determinate output, and
- they allow clear, concise, easy to understand code.

In contrast, [6, p. 490–491] imperative languages such as Fortran have a number of shortcomings for scientific parallel programming. Fortran is not very good for expressing high-level abstractions, such as abstracting behavior into a function (higher-order functions). Fortran's imperative nature forces the user to overspecify execution order, making it very difficult to compile good code for a parallel machine. Most importantly, Fortran *relies on the user for determinate programs*, instead of guaranteeing the determinacy of programs, as functional languages do.

Many of these criticisms can also be applied to other imperative languages as well. In fact, we can make a strong argument about the [21, p. 332] suitability of functional and imperative languages:

functional languages: are suitable to express equation solving, not for expressing non-determinism and mutable objects

imperative languages: are appropriate when non-determinism and mutable objects are important in the problem domain (*e.g.* in airline reservation systems, operating systems, *etc.*)

In the context of scientific parallel programming, functional languages have the advantage of expressiveness, as scientific programming is mainly about implementing algorithms, and not about

dealing with mutable objects. In effect, functional programming has given the necessary abstractions to move the programmer closer to her problem domain, and has provided logical independence by removing responsibility for the issues of partitioning, scheduling, synchronization and memory latency from the programmer.

1.4. Key Points

Let me summarize the key points I have made in this chapter.

The *problem to be solved* is finding parallel programming languages and parallel architectures to allow scientific users to apply parallel processing to intermediate-level vision research The *contribution of the thesis* will be the comparison and evaluation of three general-purpose systems (both architectures and languages) for parallel intermediate-level vision. I believe the best suited is the Id functional language on the M.I.T. Tagged-Token Dataflow Architecture, because of its mathematical flavor, general applicability, lightweight task creation, determinacy, and the logical independence it provides the programmer

I define a number of parallel processing system evaluation criteria, to evaluate the match between parallel processing systems and intermediate-level vision parallel programming needs These criteria include closeness of the parallel programming language to the problem domain, determinacy of results, logical independence, and lightweight task creation, among others. These criteria evaluate how each system solves the four crucial issues in parallel execution, partitioning, scheduling, synchronization, and memory latency, which influence a parallel program's performance and how the parallel programmer sees the parallel architecture. The considerations arising from these issues in intermediate-level vision are more difficult for the programmer to handle because of the fact that processing is data dependent, whereas in low-level vision processing it is fixed Intuitive partitioning schemes also ignore large amounts of parallelism present in intermediate-level vision algorithms.

Thus, scientific parallel programmers, and intermediate-level vision programmers in particular, cannot be expected to apply machine-specific expertise about partitioning, scheduling, synchroniza-

tion and memory latency and create programs that are determinate, let alone efficient; there must be more *logical independence* in parallel programming systems, so that these concerns be moved away from the programmer, to compilers and to the parallel architecture itself. This logical independence is provided by functional programming systems, but is not found in current imperative parallel programming systems.

2. THREE APPROACHES TO THE PROBLEM

In this chapter, I will describe the three different parallel programming systems whose suitability for intermediate-level vision parallel programming I will examine in this thesis:

- Imperative programming in C with the Uniform System library of parallel routines on the BBN Butterfly multiprocessor.
- Imperative programming in C-Linda on the Sequent Balance multiprocessor.
- Functional programming in Id on the Id World simulation of the MIT Tagged-Token Dataflow Architecture (TTDA).

I will also describe the reasons for my choice of systems.

This chapter provides three answers to the vision programmer's question, "What kind of system is available for parallel processing of vision algorithms?" Because of our focus on intermediate-level vision, the systems we have selected are general-purpose MIMD systems. In chapter 3, I will show how each system performs as an intermediate-level vision parallel processing research tool by programming a simple test algorithm on each, then looking at the evaluation criteria given in section 1.2.1. The purpose of the present chapter is to get familiar with the essentials of each system.

2.1. Uniform System Programming on the BBN Butterfly

In this section we look at programming on the BBN Butterfly multiprocessor. We will give a brief description of the BBN architecture, then examine the programming environment offered by the vendor, and most importantly the parallel programming model, the Uniform System (U.S.).

2.1.1. The BBN Butterfly Architecture

The BBN Butterfly's architecture is a *shared memory* MIMD multiprocessor computer [10]. Each node is a Motorola 68020 processor with local memory. Up to 256 nodes can be connected through a butterfly switching network, shown for 16 processors in figure 2.1. Taken together, the local memories on all nodes form a global memory space: any processor can access any memory through the network. An N processor system uses $(N \log_4 N)/4$ switches; thus, a remote memory reference is $\log_4 N$ switch hops away. This is an important point: a remote memory access takes about 4μs, 5 times as long as a local memory reference. This *non-uniform memory* architecture is carried through into the Uniform System programming system.

2.1.2. The BBN Uniform System Programming Model

The programming system supplied by BBN is the Uniform System (U.S.). The Uniform System is a library of C/FORTRAN routines which provide memory and processor management for parallel programming on the BBN Butterfly. The reader interested in more detail is directed to [11]. In this thesis, we will discuss exclusively Uniform System C programming for the Butterfly, although other parallel programming systems exist [30], because it is the system supported by the vendor, and thus an example of the state of commercial parallel programming support.

The Uniform System's memory management features are the following. The programmer can set up a shared memory space across all processor/memory nodes (collectively, all memories of the processor nodes form the shared memory). The programmer can scatter large data structures

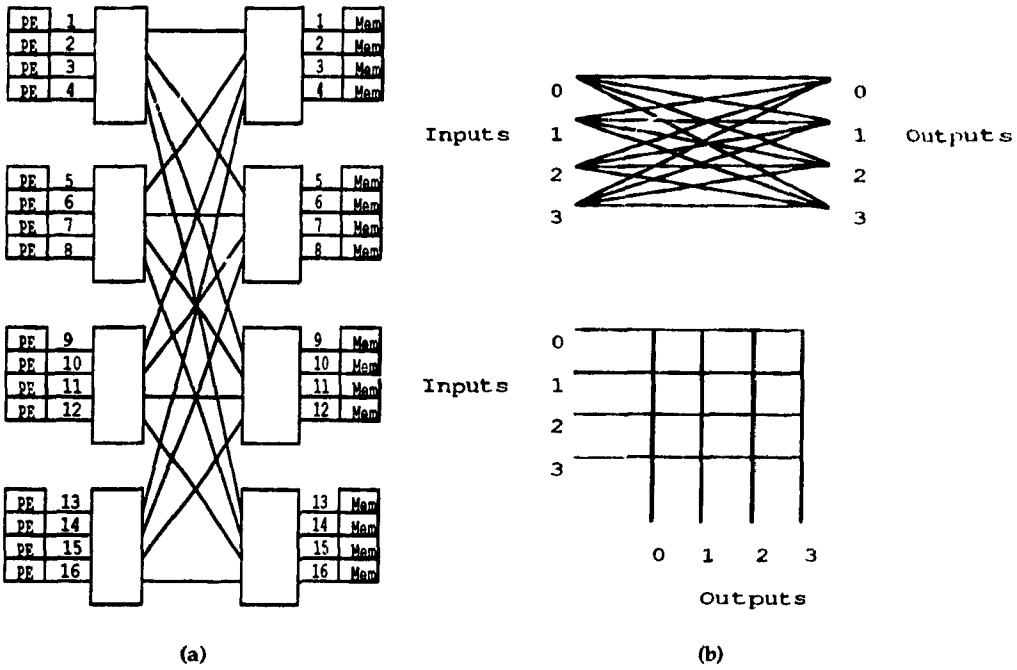


Figure 2.1: In figure (a), the BBN's butterfly interconnection network for a 16-node machine Each switching node is a 4×4 crossbar, as shown in (b) (two views of an identical crossbar) Processing elements (PE's) and memories with the same number are actually part of the same node.

across all memories, to make use of the full switching network (memory) bandwidth and thus avoid contention for a single memory. Atomic memory operations and simple spin locks[1] are provided. There are also processor management features provided. Routines exist to set up *task generators* to generate tasks for the processors: each task is a (C/FORTRAN) function (subroutine), and will be dynamically scheduled to a processor at run-time. Load balancing is thus dynamic, which can result in better processor utilization, depending on program partitioning. Each processor under U.S. management runs a single task, to avoid costly context swaps.

As mentioned earlier, the Uniform System is based on libraries of parallel routines for C-language programs. the compiler used for Uniform System programs is a standard C compiler which generates sequential code. The task generators provided by BBN are based on the model of *applying a function to each item of a data structure* (*e.g.* list, vector or array) *in parallel*. Partitioning can be based either on the output data structure (*e.g.* one task to compute each element of a matrix multiplication result), or the input data structure (*e.g.* summation of the elements of a matrix, with each task assigned a row). U.S. programming is imperative: tasks do not return values, so the model is one of applying a function to an input data structure to side-effect results to an output data structure, in parallel. This assumes independence between each task execution, and therefore forces the user to worry about data dependencies: for example, programming a matrix algorithm that proceeds along a wavefront would require some restructuring to fit the U.S. mechanism and ensure independence of the tasks. However, the generator mechanism is well-structured. For example, there are generators that operate on data structures with one and two indices (typically vectors or matrices). Used in this way, the generator mechanism is semantically close to iteration, with all iterations done in parallel. Generators can be synchronous or asynchronous, respectively returning control to the caller at the end of all tasks, or immediately. In its most general form, which is to generate tasks from a list of tasks, and used synchronously, the generator mechanism is like the `parbegin parend` construct found in some parallel languages. This kind of structure simplifies synchronization in a large number of cases, although at the cost of loosing flexibility in the task creation mechanism. It is more difficult to spawn arbitrary tasks, and far easier to fit the generator model of parallelism over a data structure: "The easiest way to achieve parallel operation is to structure the program to fit the mold of one of these task generators" [11]. We shall see that the problem with this approach is that fitting the mold often involves restructuring the algorithm to fit the available mechanisms.

[1] Shared variables that are read repeatedly and whose change of value signal a synchronization event

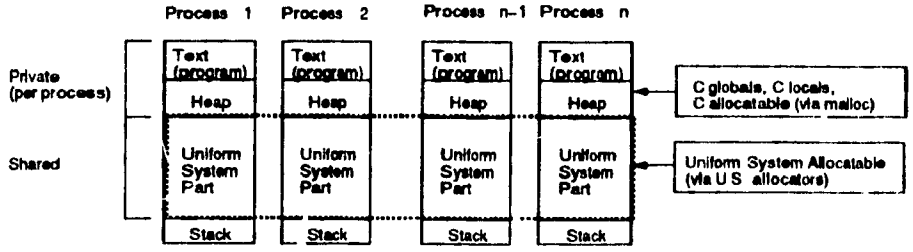


Figure 2.2: The Uniform System address space, showing the mapping of C variables and storage to actual physical storage Adapted from [11].

A very important point for the programmer to keep in mind is the machine's hierarchical memory model Physically, each node in the machine is composed of a processor and local memory; the memories at each node collectively form a shared memory, through the butterfly interconnection network mentioned in section 2.1. Access to local memory is *five times faster* than access through the network to remote memory[2]. In terms of Uniform System C programming, the distinctions between the various types of storage are shown in figure 2.2. Notice especially that C globals are process private; in fact, while such variables are stored at the same address on all nodes, changing these variables on one processor will only make the change on that processor, and on no other If a programmer wants the change to a variable to be seen on all processors, the variable must be stored in Uniform System shared storage. These distinctions are of extreme importance (and a source of programmer difficulties) in Uniform System programming.

A summary of how Uniform System C programming solves the four problems of section 1 is therefore the following. Explicit partitioning is needed; for ease of programming, the programmer should partition to fit the model of parallelism over data structures offered by the Uniform System. The match of partitioning granularity to architecture granularity is uncertain. The model for scheduling is implicit, dynamic self-scheduling. Synchronization is explicit, through the low-level mechanisms provided in the U.S. Some synchronization is simplified because synchronous task generators are provided. It is up to the programmer to specify the placement of data to solve the memory latency problem, as memory is either fast, contention-free, but private, or slow, contention-prone, but shared.

[2]Note that some shared memory is allocated locally on a processor node ($1/p$ of the total shared memory, in a p-node system) A programmer can choose to place data specifically in local shared memory (on any node, in fact). However, of course, placing data shared by many processors on a single node will cause memory contention.

An example of U.S. programming for matrix multiplication is given in appendix A.

2.2. C-Linda Programming on the Sequent Balance

In this section we look at programming with the C-Linda parallel language, running on a Sequent Balance multiprocessor. We will give a brief description of both the Sequent Balance architecture and of the C-Linda language. The reader interested in more detail on either topic is referred to [38] and [2], respectively.

2.2.1. The Sequent Balance Architecture

The Sequent Balance is a bus-based, shared-memory multiprocessor. Each processor has cache memory (write-through, with bus snooping logic[3]), *but no local memory* for user processes. Instead, there is a single global memory, and only frequently accessed, read-only kernel data is kept in a small local memory. The memory system is pipelined and asynchronous, to maximize the use of the bus. A SLIC chip (System Link and Interrupt Controller) at each processor takes care of message-passing interrupts between processors on a separate bus, and each SLIC holds copies of synchronization *gates*[4]. The Balance runs Dynix, a multiprocessor UNIX with a single process queue for all processors. As there is only one main memory, process distribution and migration is trivial, but of course at the cost of bus traffic and cache updating. Good performance is obtained on multiuser or multitask loads.

[3] A write-through cache immediately copies written data to main memory over the bus; bus snooping logic listens on the bus to check if data in a processor's cache has not become invalid because of a memory write.

[4] Equivalent to binary semaphores.

2.2.2. The C-Linda Programming Model

We first start by explaining the core of the C-Linda language, the Linda coordination language [21]. Explicitly-parallel programming has two components, computation and coordination. Coordination is composed of communication and synchronization, therefore, adding a coordination language such as Linda to a base language like C will result in a parallel dialect of the base language, in this case, C-Linda.

Linda itself is a set of 6 operators that can be added to any base language. Linda parallel tasks communicate through a shared dataspace called tuple space, regardless of whether or not the machine on which Linda is implemented has physically shared memory. The tuple space memory model is central in Linda: the storage unit is not the byte, but the *tuple*, or ordered set of values. Tuples in tuple space are accessed associatively, through a logical name, where the logical name is any selection of the tuple's values.

There are three operations on tuples in tuple space: read, add, and remove — there is no modify. This atomicity makes it possible for many processes to share tuple space and use it as a means of synchronization and communication. Data in a Linda program is never exchanged directly between two processes; instead, a process with data to share adds it as a tuple to tuple space. A process wanting to receive data can either remove a tuple from tuple space, or simply read in a copy of the tuple in tuple space. Communication between processes is therefore uncoupled, in space and time: a process does not have to know where the data is going, as it simply places it in tuple space for all to access (anonymous communication), and does not have to synchronize with the process receiving the data, which can simply read or remove the data from tuple space at any later time. Modifying data in a tuple means removing the tuple from tuple space, changing the data value, then placing the tuple back into tuple space. Thus, the semantics of the operations on tuple space allow for easy synchronization and communication.

To summarize then, the Linda coordination language is a set of 6 language-independent operators that allow parallel tasks to communicate and synchronize atomically and anonymously through a shared, associative dataspace, called *Tuple Space*.

A tuple in C-Linda could be, for example, `("a string", 15, 17.543, "string 2")`, or `(0, 1, "foo")`, or any other series of typed fields, the allowable types being dependent on the base language. The names of the operations on tuple space are `out(t)`, `in(s)`, `rd(s)` and `eval(t)`. `out(t)` causes tuple `t` to be added to tuple space. `in(s)` causes a tuple `t` that matches anti-tuple `s` to be removed from tuple space; if no matching tuple is found, the process executing the `in(s)` will block. An anti-tuple is structurally the same as a tuple, except that some or all of its fields may be formal parameters, which get bound to corresponding actuals in the matched tuple when it is removed from tuple space. For example, the anti-tuple `("bar", ?i, ?f)` matches the tuple `("bar", 2, 7.89)` (if `i` and `f` are an `int` and a `float`, respectively), since they have the same number of fields, the same actual in the first field, and matching types in the last two fields. After doing `in("bar", ?i, ?f)`, `i` and `f` would be bound to 2 and 7.89, respectively. If more than one tuple matches an anti-tuple, an arbitrary (non-deterministic) choice is made for the tuple to be removed. This tuple removal and formal assignment mechanism is shown in figure 2.3. `rd(s)` is similar to `in(s)`, except that a copy of a matching tuple tuple is returned; the tuple is not removed from tuple space. `eval(t)` is the same as `out(t)`, except that the tuple is evaluated after being placed into tuple space rather than before. `eval(t)` is thus the mechanism for task creation in Linda: it places an active tuple into tuple space, instead of a passive tuple, as does `out(t)`. When evaluation of the active tuple is finished, it turns into a passive data tuple, identical to those placed in tuple space by `out(t)`.

It is important to realize that tuples exist independently of the processes that created them, and may collectively form data structures in tuple space. For example, as shown in figure 2.4, a tuple space matrix could be a collection of element tuples. It could also have been a collection of row or column tuples, or a collection of sub-matrix tuples; the choice of representations depends on programmability and efficiency considerations, as we shall see.

A summary of how C-Linda programming solves the four problems of section 1 is therefore the following. Explicit partitioning into potentially arbitrary tasks is necessary. The match of partitioning granularity to architecture granularity is uncertain. The model for scheduling is implicit, dynamic self-scheduling. Synchronization is explicit; however, the anonymity and atomicity of tuple space operations create a powerful and flexible mechanism for synchronization. Memory latency considerations are partly up to the programmer to solve, as data placement is out of programmer control, but tuple space data structure granularity is. A finer partitioning increases parallelism, but

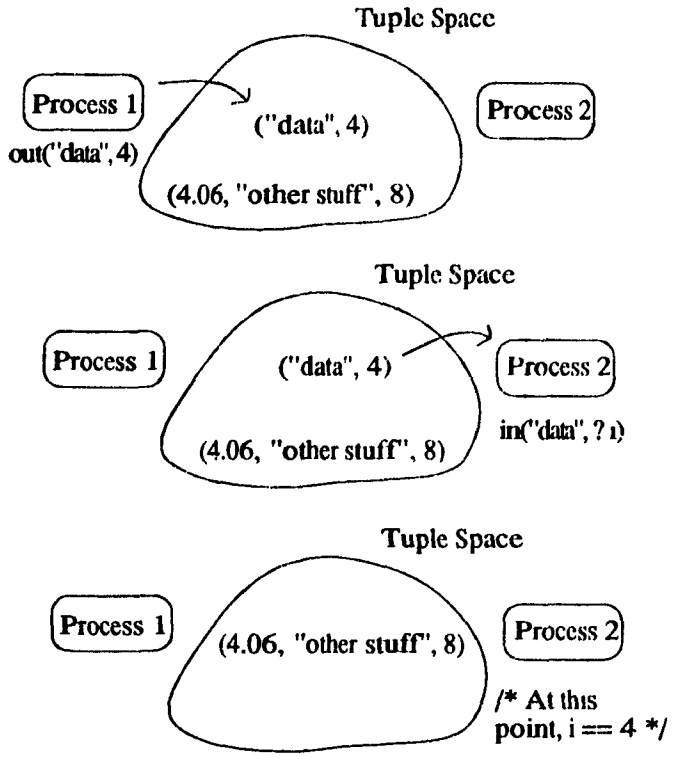


Figure 2.3: A Linda example. Processes 1 and 2 were previously created using `eval()`. The anti-tuple `in("data", ?i)` in process 2 matches the tuple placed in tuple space by process 1, and formal `i` gets bound to value 4.

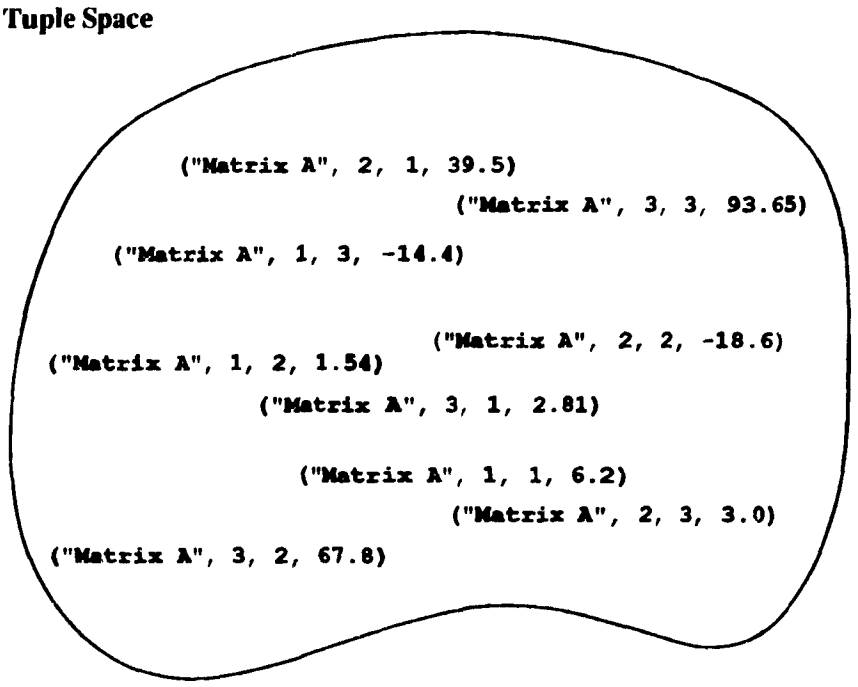


Figure 2.4: A tuple space data structure. Matrix A is stored in a collection of single-element tuples.

increases access latency, while coarser partitioning decreases latency but increases contention[5]. This issue is very architecture dependent.

2.3. Id Programming on the TTDA Simulator

In this section, we give a brief introduction to dataflow architectures, briefly describe the MIT TTDA architecture and the Id functional programming language, and examine the system on which Id code was run, a software simulation of the Massachusetts Institute of Technology's Tagged-Token Dataflow Architecture (TTDA) called GITA, part of a software development environment called Id World.

2.3.1. An Introduction To Dataflow Architectures

We will provide a brief introduction to dataflow architectures; our description follows [4], which the interested reader is urged to consult for greater depth.

The dataflow concept is quite simple: a dataflow program is a directed graph where nodes are operations and arcs denote data dependencies between operations. Data values are carried on *tokens*, which flow along the arcs. A node may execute (or *fire*) when a token is available on each input arc. When it fires, a data token is removed from each input arc, a result is computed using these data values, and a token containing the result is produced on each output arc For example, the following program is easily converted into the dataflow graph of figure 2.5

```
let x = a*b;
    y = 4*c
in  (x + y)/c.
```

[5]Which, in turn, increases latency!

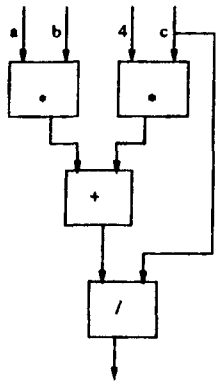


Figure 2.5: A simple dataflow program.

Note how the result of an operation is purely a function of the input values; there are no interactions between nodes via side effects, for example, through shared memory. The example shows the two key properties of dataflow architectures: *parallelism*, as nodes can execute in parallel unless there is an explicit data dependence between them, and *determinacy*, as results are completely independent of the order in which potentially parallel nodes fire. More general programs (such as those with loops and conditionals) can be created with boolean tokens and *switch* and *merge* operators.

How does a real dataflow machine execute such a program? Dataflow graphs such as the one in figure 2.5 can be viewed as a machine language for a dataflow machine, where a node in the graph is a machine instruction. Each instruction contains an op-code and a list of destination instruction addresses (for the result token). The basic instruction cycle for any dataflow machine is thus:

1. detect when an operation is enabled;
2. determine the operation to be performed (*i.e.* which op-code);
3. compute the result;
4. generate result tokens.

Note how the first stage of the instruction cycle allows us to avoid performance degradation because of memory latency (delays through network to memory): if a result hasn't arrived yet, the

operation won't be enabled, but we can simply select another one which is enabled. von Neumann architectures, with their sequential program counter, must block until the data arrives. The dataflow architecture therefore *uses parallelism* (more than one operation to be selected from) to *hide latency*.

What are the problems with dataflow architectures? Simply that the hardware cost of a single dataflow processor is much higher than that of a traditional von Neumann processor: for example, selecting the instruction to execute in a dataflow architecture is more complex than simply incrementing a program counter in a von Neumann machine. The gains to be made in adopting the dataflow architecture arise in a multiprocessor setting: because dataflow machines hide memory latency by using parallelism and switching to an enabled instruction if necessary, their processors can be busy a larger percentage of the time than von Neumann processors in a parallel machine, which can only idly wait for their operands to arive from memory after network delays or contention.

Notice now how functional languages and dataflow architectures are a good match for parallel processing. Both work on the principle of producing results from expressions and not on side effects to memory. Additionally, to be efficient, dataflow architectures need the high parallelism (to be able to switch to another instruction to hide latency) present in functional programs, which impose the minimal restrictions on parallelism (only those that arise from data dependencies). On the other hand, multiprocessors built from von Neumann processing elements and programmed with explicitly parallel languages derived from sequential languages suffer from serious problems. They impose on the programmer the need to specify synchronizations to avoid read/write races (which cause non-determinacy), and subtle timing bugs arise. Functional languages completely avoid these synchronization problems by disallowing "updatable variables" (*i.e.* side effects to memory).

2.3.2. The MIT Tagged-Token Dataflow Architecture

In this section we will take a very brief look at the MIT Tagged-Token Dataflow Architecture (TTDA). Our treatment will follow that of [7].

The TTDA consists of a number of identical processing elements (PE's) and I-structure storage elements (described in section 2.3.3), connected through a packet-switched network. The I-structure

elements collectively implement a global shared memory. A single PE-I-structure storage pair is, in itself, a complete dataflow machine [7]. In the multiprocessor configuration of the TTDA, all memories are globally accessible. Code can be distributed and executed over many PEs (processing elements) (a single code block executing simultaneously over many PEs, or even part of a single code block executing on different PEs). However, mapping over multiple processors does not increase overhead: the number of instructions executed in a TTDA program is independent of the number of PEs it is run on [7].

As in other dataflow machines, the TTDA has fast context switching and split-phase memory transactions (*i.e.* switching to another instruction if a memory transaction has not completed, as described previously), to ensure that synchronization occurs at the finest level possible and that memory latency effects are reduced.

We obviously cannot look at the MIT TTDA in great detail here; the interested reader is directed to [7]. The important point to remember is that the MIT TTDA executes dataflow graphs, obtaining maximal parallelism up to data dependencies and machine constraints. Fine-grained synchronization is supported through I-structure storage, and its mechanism of PRESENT/ABSENT/WAITING indicators (see page 42).

2.3.3. The Id Functional Language

As described in appendix B, the class of applicative, or functional, programming languages is one in which computation is carried out entirely through the evaluation of expressions (*i.e.* the application of functions to arguments, thus producing results) [24], and completely without side effects; these characteristics are found in Id. For a complete description of the Id language, see [27]. Additionally, Id fully supports higher-order functions, data abstraction, pattern matching and array comprehensions.

We present a very simple (and, for the moment, also very inefficient) functional program for matrix multiplication (figure 2.6), written in the functional language Id [27]. We rely heavily on the intuition of the reader to understand program syntax.

```
def ip C D =
  {
    (_,n) = bounds C
  in
    sum (1,n) {fun i = C[i]*D[i]}
  };

def row i E =
  {
    (_,_),(_,n) = 2D_bounds E
  in
    {vector (1,n)
     | [j] = E[i,j]  ||  j <- 1 to n}
  };

def col j F =
  {
    (_,n),(_,_) = 2D_bounds F
  in
    {vector (1,n)
     | [i] = F[i,j]  ||  i <- 1 to n)
  };


def matmult A B =
  {
    (_,m),(_,n) = 2D_bounds A;
    (_,_),(_,l) = 2D_bounds B
  in
    {matrix (1,m),(1,l)
     | [i,j] = ip (row i A) (col j B)
               ||  i <- 1 to m
                &  j <- 1 to l}
  };
```

Figure 2.6: Example functional program: matrix multiplication

Four functions, `ip`, `row`, `col` and `matmult` are defined. Respectively, these functions return an inner product, a row of a matrix, a matrix column, and a matrix product. In the code, _ is a pattern-matching character, and matches any actual value. Parentheses are used only for grouping, and can otherwise be omitted.

Certain characteristics of the program are notable In line 5, the sum function (defined elsewhere) takes two arguments:

- the first is a 2-tuple which describes the summation bounds
- the second argument to sum is itself a function of one argument; this argument is the summation index `i`.

Note how close this definition of sum is to the mathematical notation for summation,

$$\sum_{i=1}^{n} C_i D_i \tag{2.1}$$

We have abstracted out the summation behavior and put it in a function, which expects another function (the summand, here `C[i]*D[i]`) as an argument. Thus, higher-order functions raise the level of programming by making it possible to create functions that operate on other functions. Note also the *array comprehension* syntax for declaring and filling arrays in the same code fragment (see appendix B).

Id is therefore a functional language, but augmented with non-functional data structures called *I-structures* [7]. I-structures can be declared in one place and filled in at another; their name is derived from the fact that they can be filled incrementally [4]. However, to prevent the possibility of read/write races, I-structure slots have PRESENT/ABSENT/WAITING indicators, and cannot be written to more than once, thus preserving the functional nature of a program. I-structure slots with WAITING indicators have had a read attempt performed on them while the slot was empty; those read requests are deferred and stored in a part of I-structure storage specially reserved for that purpose, in the MIT Tagged-Token Dataflow Architecture [4]. As stated in [4], the main reason for introducing I-structures to Id was to obtain non-strict data structures that could later be filled in a demand-driven way, which cannot be done with array comprehensions, which, as seen above,

are declared and filled in one place. For situations in which this can be a limitation (and thus where I-structures provide a solution), see [24]. In effect, I-structures allow for very fine-grained synchronization, at the array element level. The underlying architecture must be able to efficiently support this.

Perhaps the most important aspect of Id is that it is determinate, as for all other functional languages. That is, given identical inputs, the outputs of a computation will always be the same, regardless of the order in which computations occur. This frees the programmer from the details of scheduling and from having to synchronize parallel activities; an Id program *will be* determinate, irrespective of the Id code it contains.

As mentioned earlier, Id's operational semantics also free the programmer from having to identify parallelism: parallelism in Id is implicit and compiler-detected.

To summarize then, the Id functional language provides determinacy, higher-order functions, array comprehensions and pattern matching for expressiveness, data abstraction for modularity, and I-structure arrays for fine-grained synchronization.

2.3.4. Id World, GITA, and Id Software Development

The experimental algorithm implementation described in section 3.1 was done on a SparcStation running Lucid Common Lisp, on which is implemented the Id software development environment and MIT TTDA software simulator, Id World and GITA, respectively.

Id World is an integrated software environment that features an editor, a software emulator of the TTDA called GITA (with debugging support), and extensive performance and monitoring tools [29]. Code execution, debugging, and statistics monitoring occurs in GITA, the Graph Interpreter for the Tagged-token Architecture.

Statistics collection allows the user to collect statistics about the parallelism profile of an algorithm, the mix of instructions executed, various I-structure storage operations, *etc.* Different emulation

modes allow the user to change the number of processors from infinite to finite, and communication latency (explained in section 3.6) from 0 to a non-zero value.

A summary of how Id programming on the TTDA solves the four problems of section 1 is therefore the following. Partitioning is implicit (compiler-determined) Each instruction is a task. Scheduling is also implicit, performed at run-time on the TTDA. Synchronization is implicit and fine-grained, either through the implicit ordering of the execution of individual instructions, or through accesses to I-structures. The memory latency problem is solved by the architecture through split-phase memory transactions.

2.4. Reasons for Experimental System Choices

As mentioned in chapter 1, our goal for the research was to investigate general purpose systems for the vision research environment, appropriate for intermediate-level vision. We felt that this excluded systems which only support message passing programming models, as their lack of support for globally-shared data objects (*e.g.* input iconic data, in the case of intermediate-level vision) is a heavy burden on the programmer for placement and movement of data and tasks.

An interesting issue that we wanted to investigate was that of imperative parallel programming versus declarative parallel programming, which are two camps in parallel programming community. Imperative parallel programming advocates emphasize the ease of learning parallel constructs for a well-known sequential language, giving explicit programmer control for presumably better performance, or the convenience of automatically parallelizing sequential language programs. Declarative parallel programming researchers emphasize the importance of determinacy, the high degree of parallelism exposed by functional languages, the programming expressiveness provided, and the logical independence from machine-dependent issues. In fact, we can see that the parallel programming systems chosen support different levels of logical independence:

Id: implicit partitioning, synchronization, scheduling

C-Linda: explicit partitioning, simplified (but explicit) synchronization, implicit scheduling

BBN Uniform System: explicit partitioning, explicit synchronization, implicit scheduling

2.5. Key Points

Let me summarize the key points I have made in this chapter.

The BBN Butterfly is a shared memory multiprocessor with a high-bandwidth, non-uniform memory architecture; the BBN Uniform System (U.S.) parallelism model is based on side effects to data structures. Programmers must structure their program accordingly, and worry about synchronization between tasks.

The Sequent Balance is a single-bus, shared memory multiprocessor. The C-Linda parallelism model is based on arbitrary task creation and anonymous, atomic operations on a shared, associatively-accessed dataspace.

The MIT Tagged-Token Dataflow Architecture (TTDA) is a dataflow multiprocessor that supports fast context switching and I-structures for fine-grained synchronization and split phase memory transactions to reduce memory latency effects. It is simulated by the heavily-instrumented GITA, the Graph Interpreter for the Tagged-token Architecture. Id is a modern functional language augmented with I-structures to allow for very fine-grained synchronization.

The above three parallel processing systems were chosen because of general applicability, and contrast between explicitly parallel imperative languages and implicitly parallel functional languages.

3. A COMPARISON IN THE CONTEXT OF INTERMEDIATE-LEVEL VISION

3.1. An Intermediate-Level Vision Example: Parallel Cooperative Fitting

In this section we will describe an experimental intermediate-level vision algorithm we will use to evaluate our three parallel processing systems. It is a parallel cooperative fitting algorithm: multiple fitting processes cooperate and exchange information to change the result of each fitting process. Our work on this algorithm is incomplete; indeed, the algorithm itself may need a great deal of refinement. However, this is typical of research work and will be useful in evaluating parallel processing systems. We will describe the algorithm in some detail, explain why it fits the criteria detailed in section 1.1.2, and what makes its implementation interesting.

3.1.1. The parallel cooperative fitting algorithm

The idea behind this demonstration experimental algorithm is to use additional information present in a data set with multiple components to constrain a fitting procedure applied to each component. These constraints are communicated between fitting processes while the fitting takes place. In essence, the goal is to perform, in parallel, model fitting of n volumetric primitives and m surface curves (either bounding contours or *inter-penetration curves*) to a segmented (range) image. Each fitting process is iterative: a process moves one step toward what it believes is the correct fit, given

the information it possesses. It then exchanges information with its neighbors, and takes another step, which presumably is an improvement upon the previous one. This iterative procedure continues until some measure of convergence is reached.

The vision processing task of interest in this case is the fitting of volumetric primitives to (three-dimensional) range data, so as to infer volumetric models from dense three-dimensional input [39]. However, previous approaches use only part of the information present in the input data, namely the range information. The rationale for using parallel cooperative fitting is to use additional information present in the range image in the fitting process, such as bounding contours and inter-penetration curves. An initial estimate must be done for these and for the volumetric models before starting the fitting algorithm (*e.g.* by performing a least-squares fit for the volumes).

The 3D bounding contour of the range data could be determined by first removing background points, then considering only those data points whose surface normal falls beyond a specified threshold ϵ. Additionally, given a segmentation of an object composed of multiple parts, adjacency (spatial) relationships between volumes fitted to each part can also be used as a further constraint, in the form of *inter-penetration curves*, formed at the intersection of two volumetric primitives. A part could therefore have, in theory, any number of inter-penetration curves defined between it and its neighbor(s).

For example, after scanning a 3-D object to obtain range data, having segmented it into two parts, and having determined its bounding contour, we could run the parallel cooperative fitting algorithm on the data to obtain a volumetric description of the scene. A number of iterative fitting processes would be active, and communicating constraints on each other's fit.

- Two fitting processes to fit the 2 volumetric primitives to the surfaces.
- Two fitting processes to fit the bounding contours of the actual data to the bounding contours of the volumetric primitives.
- One fitting process to fit the intersection curve formed by the two volumetric primitives to the curve in the actual data, formed by finding points of extremal negative curvature, for example.

The desired result of running the algorithm in [39] on a segmented image of a toy wooden doll is shown in figure 3.1, with the additional information that could have been used, such as the bounding contour and the inter-penetration curve.

Therefore, the constraints used in this case are that the volumetric primitives should be close to the surface data, bounding contours should be close to those in the data, and curves of intersection should be close to those in the data. During the iterative fitting (minimization), fitting processes will communicate between each other, to try to satisfy these constraints. The way in which these constraints are incorporated into the iterative minimization is through a distance measure: the distance between parameter vectors in parameter space is blended into the fitting error metric. We will examine this further on.

To summarize then, range image parallel cooperative fitting can be understood in terms of the following input and output data structures. The input is a segmented range image, which includes points on the object surface (for each part), points on the bounding contours of each part, and possibly, points on an inter-penetration curve(s) a part shares with its neighbor(s). The output is a list of volumetric primitive parameter vectors (and possibly vectors describing bounding contours and interpenetration curves), one vector per segmented part (or curve).

To simplify the experimental algorithm, it was decided to solve the problem for the simple case of an ellipsoid, centered at the origin, with a known orientation, and with its bounding contour in the x-y plane. Thus, we have a very simple case of the parallel cooperative fitting algorithm, a single part and its bounding contour, and we will have only two fitting processes: a surface data *ellipsoid* fitting process, and a bounding contour (of surface data) *ellipse* fitting process. The fact that we don't determine pose or position (*i.e.* assume ellipsoid centered at origin, no translation and rotation) is not restrictive, as it only implies an extra 6 parameters to determine. We will assume that the bounding contour lies in the x-y plane, which is obviously not necessarily true These assumptions do simplify the interactions between the fitting processes a great deal.

In fact, the algorithm described below is only one possible implementation. The assumptions we have made leave many unanswered questions as to the performance or even the applicability of the algorithm on real data, but the important point is that this could be a plausible first step in the development of a working algorithm. The worth of the approach, in terms of providing good

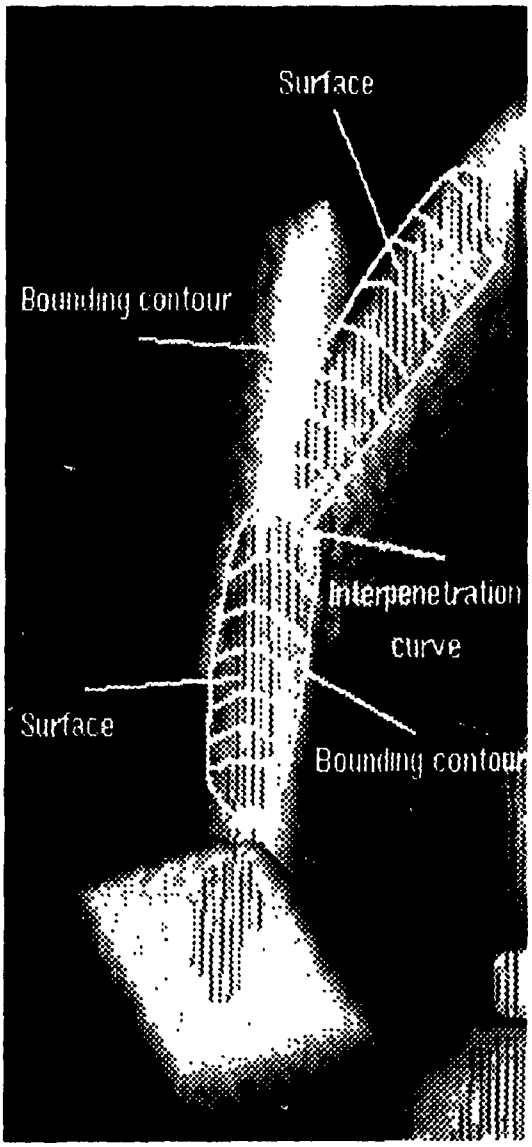


Figure 3.1: Results of running the fitting algorithm in [39] on range data from the arm of a wooden doll. The original data is shown as darker vertical bars. The grey volumes are fitted superquadric models, which are seen to overlap and exceed the bounds of the input range. The additional information that could have been used, namely the bounding contours and the inter-penetration curve, are shown in white.

computer vision results, will not be commented on further. Our interest is that the computation patterns of the algorithm feature some of the characteristics to be expected in intermediate-level vision algorithms

The full algorithm for the case of a single ellipsoid could be the following. It is based on that described in [39]. There is an iterative fitting process for the surface, and one for the bounding contour. Independently, each would produce a parameter vector. Both should be describing the same object, but because of noise in data acquisition, they will differ slightly. Correspondingly, at each iteration step, we try to minimize the difference between the parameter vectors produced by fitting to the surface and by fitting to the bounding contour. We do this by defining a χ^2 merit function to be minimized that incorporates a function of the distance between the 2 parameter vectors in parameter space. Iterative minimization is necessary because of the non-linear dependence of the χ^2 merit function on parameters. The algorithm we use is *Levenberg-Marquardt* iterative minimization [33] (which continuously changes from gradient descent far from the minimum to an inverse Hessian "jump" when closer to the minimum).

Any fitting process is obviously dependent on the error metric used. The error metric for the ellipsoid can be derived from the equation of the ellipsoid. We can consider the left-hand side of this equation as an *inside-outside* function (w.r.t. the RHS, which is 1). Thus, the difference between $f(\mathbf{x}, \mathbf{a})$ ($\mathbf{x}$ is data, $\mathbf{a}$ is the parameter vector) and 1 is an error metric (the D_1 metric [39, p. 691]). Thus, for the ellipsoid,

$$D_1(\mathbf{x}_S, \mathbf{a}_S) = f(\mathbf{x}_S, \mathbf{a}_S) - 1 = \left(\frac{x_s}{a_s}\right)^2 + \left(\frac{y_s}{b_s}\right)^2 + \left(\frac{z_s}{c_s}\right)^2 - 1 \tag{3.1}$$

where $\mathbf{x}_S = (x_s, y_s, z_s)$ is a surface point, and $\mathbf{a}_S = (a_s, b_s, c_s)$ is the surface parameter vector.

To be able to combine the two parameter vectors in this error measure, we will add in their difference in the error measure, as follows:

$$\left(\frac{x_s}{a_s}\right)^2 + \left(\frac{y_s}{b_s}\right)^2 + \left(\frac{z_s}{c_s}\right)^2 - 1 + \lambda((a_c - a_s)^2 + (b_c - b_s)^2, \tag{3.2}$$

where $\mathbf{x}_C$ and $\mathbf{a}_C$ are the bounding contour equivalents to $\mathbf{x}_S$ and $\mathbf{a}_S$, and use that in the iterative fit. Notice that this introduces an adjustment "knob" λ, and leaves c_s specified only by the surface data,

as $\mathbf{a}_S = (a_s, b_s, c_s)$ has three components (a surface parameter vector), and $\mathbf{a}_C = (a_c, b_c)$ has only two (a curve parameter vector).

3.1.2. Relevance of the Experimental Algorithm

The algorithm is interesting in many ways. It requires a large amount of iconic input data: each fitting process will fit to a large number of Cartesian data points (each of which has three components). The output is a list of features, in this case ellipsoids and ellipses describing surfaces and curves, respectively, and each feature is described by a vector of parameters. The algorithm is numerical, involving a large number of summations (to compute the merit function χ^2, for example). Communication between tasks can be data dependent, as it depends on how many inter-penetration curves exist for a given volume. In our simplified example, however, communication is fixed, and only takes place between the fitting process for the surface and the fitting process for the bounding contour curve. Synchronization is fairly simple and data-independent in our simplified example. each iteration of the fitting procedure is performed in step.

The most interesting characteristics, however, are that the input data sets vary in size, and thus the feature computation times can be vastly different. Thus, a simple one task per feature partitioning scheme would be ineffective. Additionally, there is significant parallelism within each feature computation, as summations (the most important operation in our algorithm) can be done in parallel. Memory latency considerations are the second important characteristic. the large input data sets must be easily accessed and manipulated by parallel tasks.

3.2. Comments on BBN U.S. Programming

How does U.S. programming on the BBN Butterfly fare? It suffers from a lack of expressiveness and from a lack of logical independence, partly because of the C language on which the U.S. is based, but also because of its parallel library type of design. The C language, in some respects, does not even provide logical independence to the sequential programmer; for example, explicit

memory allocation can be tedious and error prone (*e.g.* writing to a data structure for which no memory was allocated) The U.S. programmer naturally inherits these concerns. Uniform System BBN programming suffers from the same lack of logical independence exhibited by other explicitly parallel programming systems (in terms of explicit synchronization, for example), but four problems in particular stand out

First is a *lack of higher-order functions*, which are crucial for expressive scientific programming (a problem inherited from the C language). Second, an *inexpressive parallelism mechanism*: task generators based on library routines are awkward to use, and their parallelism model of side effects on data structures is not always appropriate. Third, *explicit partitioning* is also a weakness: it is unclear *a priori* whether or not the partitioning chosen by the programmer is correct for the architecture, or is too fine, or too coarse. Finally, the *hierarchical memory model* shown by the Uniform System relies on the programmer to solve the memory latency problem by placing data in either local storage or remote storage This is a burden to the programmer for program performance and especially correctness.

The system also has certain strengths. First is *comparatively small grain size*, which helps to extract more of the parallelism present in an algorithm. Second, if tasks are small enough, and there are enough of them, *load balancing* will occur dynamically. Third, the system offers *large bandwidth to shared memory*, as the interconnection network to the shared memory will allow for high transfer rates, if data and access patterns to memory are both well distributed. Finally, a *single address space* for shared data simplifies programming, and especially inter-task communication.

Let us examine U S programming on the BBN Butterfly with respect to our evaluation criteria.

One important consideration is that there are no higher-order functions in Uniform System C programming, which of course is inherited from the C language. This forces the use of clumsy function pointer and argument list pointer passing, and also prohibits function composition. For example, we often need to pass 2 functions `A()` and `B()` to a function `C()`, and inside function `C()`, create a new function `f(A, B)`, which would then be used in yet another function. The absence of this feature in U.S. C programming means one can't create arbitrary combinations of the functions `A()` and `B()`. Any such combination, such as `f()` in the example above, must be defined in advance. Functions are the at the core of mathematical programming; any system that supports

scientific programming must simplify the creation and handling of functions; the Uniform System, being based on the C language, does not.

Of course, given enough work, U.S. programming on the Butterfly can be used for any scientific problem, because of its MIMD architecture and general-purpose processors It is not restricted to near-neighbor communication algorithms, as communication is through shared memory, which is entirely accessible to all processors. Nor to is it restricted to algorithms with very large grain sizes: once a Uniform System task generator is set up (which can be expensive), the overhead for task creation is approximately equivalent to that for a function call.

One of the main problems with Uniform System parallel programming is the inexpressive parallelism mechanism. With the U.S. generator mechanism, code must be restructured to fit the mechanism, which allows you to pass as parameters to the newly created task only a single pointer (and possibly one or two indices to indicate the task's number). This is what [32] refers to as *syntactic* crudeness. Generators are like the Lisp map function, in that they are meant to apply a function *over an input data structure* (such as an array or a vector), in parallel, and perform side effect on a result data structure. This is quite useful for a number of problems, but not for all. The generator mechanism can be made to handle any case, but with some restructuring of a user's code This is what Pancake [32] refers to as *semantic* crudeness.

The Uniform System's model of explicit partitioning of tasks and data (hierarchical memory model) and explicit synchronization are an added programmer concern. Scheduling, however, is implicit (dynamic self-scheduling).

Task partitioning is difficult on the BBN Butterfly, because of granularity considerations. It is difficult to find optimal grain size for tasks: the grain size must be small, to do dynamic load balancing and avoid idling, but must be large enough to avoid excessive overhead. If the proper grain size is not chosen, the risks are

task starvation: granularity too coarse, too few tasks, and many processors idle at the end.

excessive overhead: granularity too fine, too many tasks, and the computation/overhead ratio too low.

In fact, the approach suggested by BBN is of restructuring code after trial and error [11, pp. 3–4]:

> If necessary, it is usually relatively easy to combine small tasks at a later stage into larger, more manageable sizes; it is often more difficult to divide a task at a later stage into smaller ones.

In our parallel cooperative fitting code, there is one single kind of task that is used throughout the code, a parallel summation task, so that it was fairly easy to contain granularity considerations into a single variable. This will not necessarily be the case for all codes, of course, so that restructuring to achieve proper granularity will probably be far more difficult.

Large, shared data structures inevitably produce the memory latency problem, which, in the U.S., is partly the programmer's responsibility to solve. This shows up as tedious data partitioning. For good performance, the programmer must know about the machine's memory model, the C storage model, and structure the program accordingly. This is because parallel programming on the BBN is fundamentally different from sequential programming: the programmer must *always* keep in mind the particular storage details of each variable, which is obviously difficult to do, because there are n *different* address spaces for n processors. To the programmer, memory latency appears in two different guises:

memory contention: Data in the same memory node will cause contention.

remote memory accesses: Since local accesses are faster, one must always determine if a variable should be stored locally or remotely. Remote accesses are slower and increase contention, but if some data will be needed by all processors, then it must be placed in shared memory, which is remote to all processors but one.

It is important to realize that code which affects these two considerations is *distributed throughout the entire program*, and is *entirely the responsibility of the programmer*, who must be constantly attentive to the storage model. There is no syntactic difference between pointers to shared data and pointers to private data, thus creating a *non-homogeneous namespace*.

Because of the relative speed of local and remote memory, the programmer must often make copies of frequently-used data in local memory. The Uniform System provides a mechanism to automatically make local copies of specified data into each node's local memory before the node starts working on tasks from a generator: this is the `Share()` mechanism. Explicit `Share()`'s are a programmer-friendly way of making explicit local copies; the programmer must keep in mind that this mechanism must be reserved for read-only data, as any modification to local data is seen only locally and is unknown to the `Share()` mechanism. Aside from the `Share()` mechanism, *data must often be explicitly moved from remote memory to local memory if good performance is to be attained* local copies for data that vary are best handled explicitly by the programmer, which unfortunately is an extra worry.

Memory contention is another programmer concern. Efficiency considerations (avoiding hot spots[1]) impose data structures on the programmer. For example, C vectors are stored in a single memory node, to have vector elements in a contiguous address space (as required by C). To avoid contention for a single memory, programs which deal with long vectors must have a special vector data allocator and access mechanism, to scatter vector element storage across all memories and access them as such. Other data structures can similarly be partitioned and scattered, but not with standard U.S. functions, which only provide functions to scatter matrices. *An allocator and access mechanism must be constructed for each new data structure.*

The hierarchical memory model of the machine also reduces modularity. A module is a self-contained and discrete part of a larger program, which accepts input that is well defined as to content and structure, carries out a well-defined set of processing actions, and produces output that is well defined as to content and structure. Modularity is achieved when interactions between parts of a program or system can be rigidly restricted to the interactions between modules, which greatly simplifies the understanding of how a program works [34, p. 996]. In the Uniform System, constraints are now not only content and structure, but also location in memory: a function `a()` which creates parallel tasks to operate on its arguments must require that its arguments exist in shared memory. This constraint of location in memory propagates upward to all functions who call `a()`, and so forth. One would want to define a function solely in terms of input data and output results, but this ignores the added consideration of location in memory: a function which works in

[1] Regions of shared memory with high access frequency, and thus high contention

parallel will produce correct results only for arguments in shared memory

The non-homogeneous Butterfly namespace is a constant source of programmer concern. *Any* parameter passed by reference (*i.e* pointers) is a potential problem — does the pointer point to process-private on the local processor, or to shared (most probably remote) memory? If the pointer points to process-private memory, only the local processor will see the results of modifying that memory location, which sometimes isn't the intended behavior. This is a very important problem, as passing arguments by reference and returning altered values is a common programming model in C, especially for large data structures for which copying would be wasteful If the data is written to by many processors, it must be in shared memory, so that a pointer to it will be valid on all processors. If it is read-only data and is frequently used, performance considerations dictate that a local copy be made. In short, in a system with hierarchical memory under programmer control, the programmer must always be concerned about the questions: "Will this data be written to by many processors?" and "Where does this data reside in physical memory?" The latter consideration is not unlike message passing, except the situation is much simpler in shared memory environment.

Because of the shared/private distinction, viewing parallel programming with the U.S. as extended sequential programming is very misleading. A routine that was developed sequentially often will not work when run in parallel, because of side effects. Side effects to local memory will cause incorrect behavior when run in parallel, as the side effects will only be seen by one processor, not by all processors, which is usually the intended behavior. Thus, determinacy in the U.S. is entirely up to the programmer. It is fairly easy to write a non-deterministic program: for example, a program which creates tasks that side-effect C global variables will run correctly on one processor, but will not work on 2 or more processors, as each processor will side-effect its own copies of the C globals.

Because communication is done through the shared memory, and all processors are equidistant from the shared memory, communication is uniform. Synchronization, when necessary, must be done through low-level primitives such as atomic operations or spin locks, although higher-level constructs such as semaphores and monitors can be built from these. Unfortunately, a single, frequently-changing, and frequently accessed variable will be a problem for program efficiency on an architecture not tolerant of latency such as the Butterfly, as it is stored in a single memory (which causes memory contention and thus latency), and processors must synchronize to change it (which

causes synchronization latency).

3.3. BBN U.S. Experimental Results

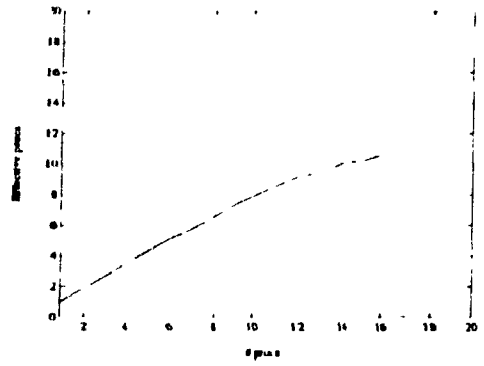
The parallel cooperative fitting code on the BBN Butterfly was parallelized simply by decomposing the summation function into a parallel one, since summation over an index range is by far the most important type of operation in the fitting code, and the easiest to parallelize. If the summation range is n, our code allows the measurement for various summation task granularities by dividing up the work into m-sized chunks.

The experiments we ran were the following. 6173 points were sampled on an artificially-generated ellipsoid, with added gaussian noise. Tests were made on a single iteration of the iterative Levenberg-Marquardt fitting procedure. Tests were run for granularities of 10, 20, 30, 40, 50, 75, 100, 125, 150, 175, and 200-point summation sub-ranges.

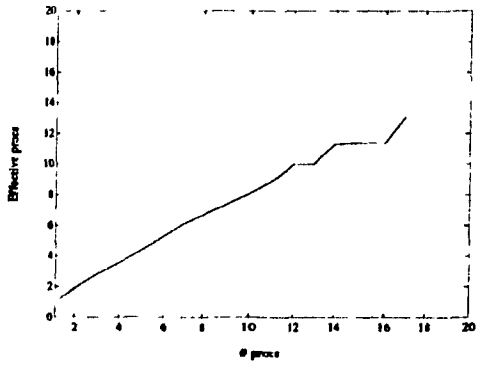
Speedup and utilization results for this test setup are shown in figures 3.2 and 3.3.

As can be seen from figure 3.2, the best speedup was obtained for a granularity of 175, which is rather coarse for the BBN Butterfly. However, the reader has certainly also noticed the "sawtooth" pattern of the speedup curve for the granularity of 175, in figure 3.2 These two observations arise because of a granularity mismatch of the program to the architecture. In the first case, better performance is obtained through coarser granularity by increasing the computation to overhead ratio.

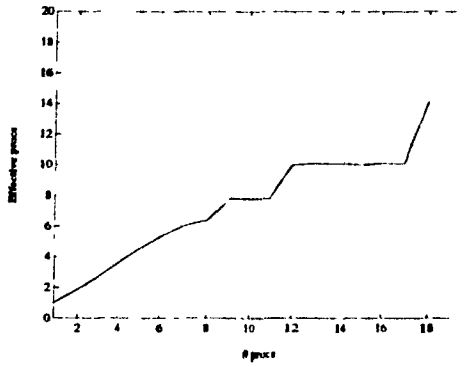
In the second case, task starvation [11] shows up as a sawtooth pattern superimposed on a generally monotonically increasing speedup curve. This can easily be seen by remembering that there are 6173 data points in the data set. A summation over 6173 data points, divided into ranges of 175 summation index values per task, gives $\lceil 6173/175 \rceil = 36$ tasks. Therefore, we must divide this number of tasks by the number of processors available. However, tasks are discrete objects, so that if we have p processors available, $p - 1$ of them will receive $\lfloor 36/p \rfloor$ tasks, and 1 processor will



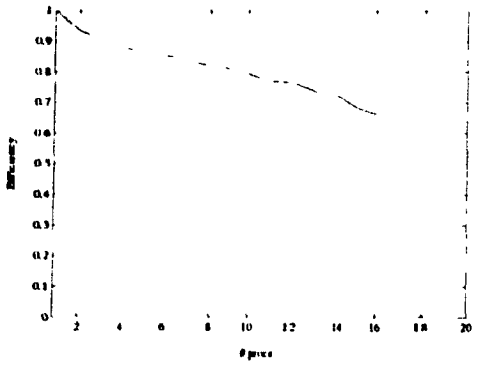
(a) Speedup, granularity = 10



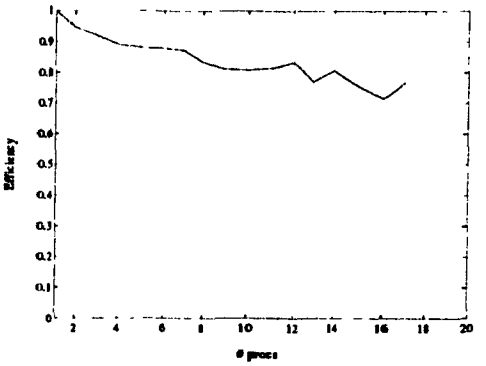
(b) Speedup, granularity = 75.



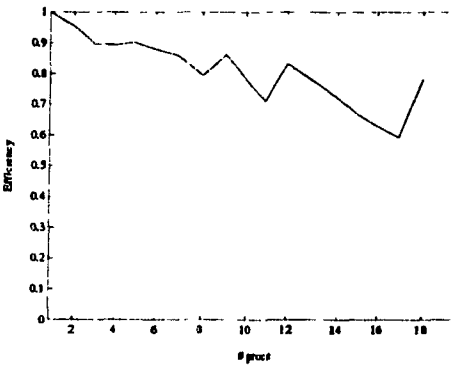
(c) Speedup, granularity = 175.



(d) Utilization, granularity = 10



(e) Utilization, granularity = 75.



(f) Utilization, granularity = 175.

Figure 3.2: Speedup and efficiency (utilization) for the BBN Butterfly, at different granularities

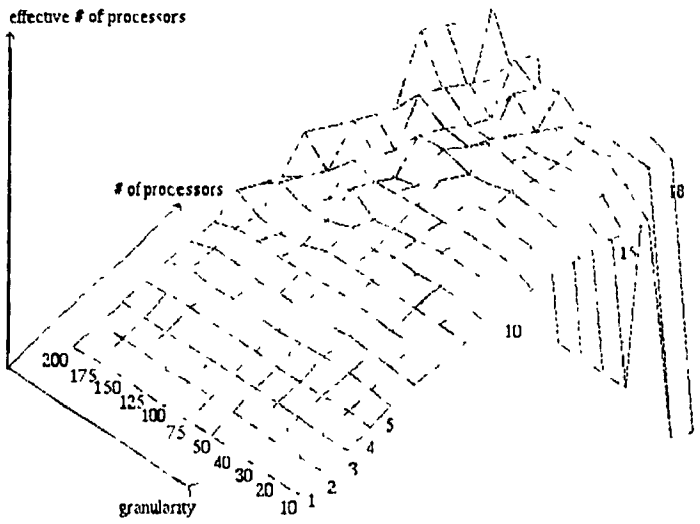


Figure 3.3: Speedup surface for the BBN Butterfly. The speedup surface shows speedup results for granularities of 10, 20, 30, 40, 50, 75, 100, 125, 150, 175, and 200, from front to back. Notice how performance improves for higher (coarser) granularities, but only for certain numbers of processors. See the text for explanations.

receive $\lceil 36/p \rceil$ tasks, so that the critical path length is $\lceil 36/p \rceil$. Assuming complete parallelization (*i.e.* ignoring sequential components of the code), speedup will therefore be $36/\lceil 36/p \rceil$. This is plotted in figure 3.4, which the reader can see compares quite well to the graph in figure 3.2, for a granularity of 175. Of course, the idealized model ignores the sequential component of the code. This includes sequential elements in both the user code, and sequential regions through the task generators.

Thus, the optimal granularity is the one which is the largest possible, to avoid overhead, while avoiding task starvation effects, to keep all processors fully utilized. This represents a slice through the speedup surface shown in figure 3.3: for a given number of processors (shown as `# of processors` in the graph), there corresponds an optimal granularity (`granularity`) at which the speedup (`effective # of processors` in the graph) is maximized Such a slice, for $p = 10, 12, 14, 16$, is shown in figure 3.5.

In general, a user will not have such a trivially simple way of changing the partition size of his or her tasks. In the vision context, dynamically generated tasks, varying data sets and processing time will play havoc with this endeavor. This will usually mean tedious restructuring for the programmer,

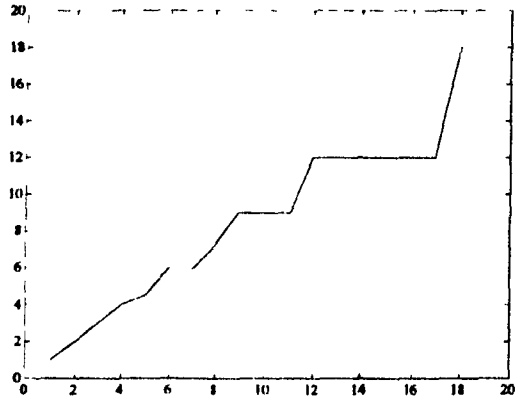


Figure 3.4: Idealized speedup curve for a granularity of 175. Note the close agreement with the curve in figure 3.2

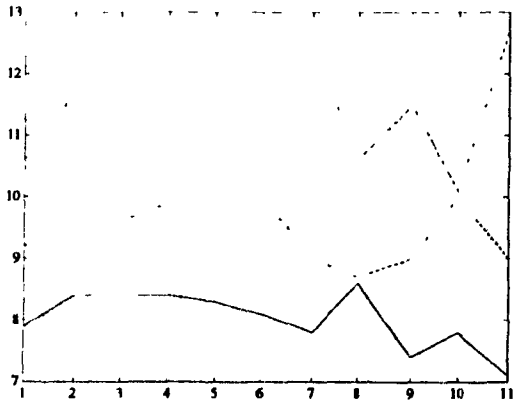


Figure 3.5: The optimal granularity for the BBN processor, for $p = 10, 12, 14, 16$. The x-axis corresponds to which of the 11 granularities we measured, from 1 to 11 respectively 10, 20, 30, 40, 50, 75, 100, 125, 150, 175, and 200. For the four cases, the best granularity is 125 (8th value), 75 or 175 (6th or 10th value), 150 (9th value), and 200 (11th value), respectively.

after much trial and error.

The final, and crucial, aspect of performance on the BBN Butterfly is memory contention For the test algorithm we used, data for fitting is most conveniently structured in the form of a vector, to be able to index through the data during summation. Unfortunately, the U S. stores C vectors in a single memory, which causes serial access to data. Therefore, a substitute data structure (a "distributed vector", distributed across the memories of the machine) had to be build for this application, which syntactically allows the programmer to access the data through a single index, but which is actually built out of the matrix scattering routines provided by BBN. This was fairly simple to do, but illustrates the point that to provide parallel access to data on the BBN, when using data structures other than simple matrices, new data structure allocation and access routines must be built by the programmer to properly scatter data across memories and obtain good performance.

3.4. Comments on Sequent Balance C-Linda Programming

C-Linda's great strength as a parallel programming system is derived from the anonymity and atomicity of the operations on tuple space. The interactions with tuple space are also a source of weakness, in some respects. The most important problems are the following. First, a *lack of higher-order functions*, crucial for expressive scientific programming (a problem inherited from the C language). Second, *explicit partitioning* is again a problem, as it is unclear *a priori* whether or not the partitioning chosen by the programmer is correct for the architecture, or is too fine, or too coarse. Third, *explicit data partitioning* is necessary To remove contention for a single data object (*e.g.* a matrix), the programmer must choose a granularity with which to decompose these objects into smaller pieces, running the same risks as for task partitioning. Fourth, there is a *loss of program modularity*, as the flatness of tuple space makes possible the anonymous synchronization mechanism of C-Linda, but also introduces the possibility of arbitrary interactions between program modules. Finally, there is *mandatory, explicit copying of shared data*. The tuple space storage model necessarily involves copying shared data from tuple space before it can be processed This can be expensive for large data objects.

The C-Linda system also has some great strengths. First, Linda offers *uncoupled, atomic operations*, as the semantics of the Linda operations on shared data guarantee atomicity, and allow an uncoupled style of programming. Second, the Linda `eval()` operation can be used to create arbitrary tasks, giving *flexible task creation*. Finally, if tasks are small enough, and there are enough of them, *load balancing* will occur dynamically.

Let us again compare with our evaluation criteria.

As for the other C language-based system we have investigated, the need for explicit memory allocation and the lack of higher-order functions in C is a serious deficiency; the same arguments as in section 3.2 apply.

An even more serious handicap arises from the following. As the reader may recall from section 3.2, C language function pointers, although they cannot be used to create new functions at run-time, at least allow the programmer to pass functions as arguments to other functions. However, only data values are allowed in tuple space, pointers to objects are not. This is logical, as Linda must be implementable on disjoint-memory machines, where pointers are meaningless from one machine to the other; shared pointers to objects only make sense in the case of a physically-shared memory. The consequences of this are obviously that you cannot share pointers to data structures; in any case, this runs contrary to the Linda tuple space model of associative storage. A more serious consequence is that you cannot pass pointers to functions in Tuple Space. Unfortunately, in C, a function is less of a name than a memory location (the value of a pointer), which, of course, in a disjoint memory environment, is completely meaningless. The same functionality can be implemented differently in C-Linda, but it is clumsy and involves more restructuring [26]. Supporting function pointers would probably involve some modification to the C-Linda implementation.

The same comments as those for the BBN Uniform System apply for C-Linda programming: C-Linda can be used on asynchronous parallel machines to program virtually any task, including intermediate-level vision algorithms.

C-Linda is not constrained by a library-based parallelism mechanism: the `eval()` operation can be used to create parallel tasks in an arbitrary way. In this sense, it is less disciplined than the U.S.

generator mechanism, but is more flexible.

The Linda model of parallelism belongs to the imperative family of side effects on data structures. `eval()` never returns a value its definition says it becomes a passive data tuple in tuple space upon completion, in effect "returning" a value through a side-effect into tuple space This is a good idea for creating distributed data structures [14], but if only a single return value is needed, it creates unnecessary access to tuple space (*e.g* an `in()` is needed afterwards to get the value produced by `eval()`). Thus is because `eval()` combines the functions of task creation and tuple space side effecting into one.

C-Linda requires explicit partitioning of tasks and explicit synchronization; however, scheduling is implicit (done at run-time). Explicit partitioning is still a burden on the programmer, in terms of finding the correct granularity, as we shall see below. Synchronization is also explicit, so that the burden for determinacy is again on the programmer; however, C-Linda's tuple space operations facilitate explicit synchronization.

Explicit task partitioning with C-Linda is difficult, and is compounded by the fact that C-Linda is portable; as C-Linda supports both shared memory machines and disjoint memory machines, a granularity that is appropriate for one architecture will not necessarily be appropriate for another For any given architecture, it is difficult to find the optimal grain size: the grain size must be small, to do dynamic load balancing and avoid idling, but must be large enough to avoid excessive overhead. This optimal granularity will vary from one Linda implementation to the other. The same problems of task starvation or excessive overhead will appear, if task granularity is too coarse or too fine, respectively.

In fact, [14] presents three parallel programming methods, applicable to other parallel programming languages than C-Linda, but well-supported by C-Linda, and show well-defined relationships between the three. Their solution to granularity problems is similar to that suggested by [11] for the Uniform System: if necessary, restructure the code to fit the architecture's granularity [14, p. 23]:

> We start with an elegant and easily-discovered but potentially inefficient solution using live data structures, move on via abstraction to a more efficient distributed data struc-

ture solution, and finally end up via specialization at a low-overhead message-passing program

For example, the goal for C-Linda parallel cooperative fitting code was to have a single summation function to do summation of any summand function in parallel. In theory, this could be done by doing an `eval()` for each invocation of the summand function, but that would be excessively fine-grained Therefore, we *restructured our program because of partitioning considerations:* because the initial granularity was too fine, we created functions to call the summand function a specified number of times. Following this restructuring, as for the BBN Uniform System code, granularity considerations in the parallel cooperative fitting code were fairly easy to restrict to a single variable, because of the structure of our demonstration problem.

As for the U.S., a single, frequently-changing, and frequently accessed variable will greatly affect program efficiency on a latency-intolerant architecture, as it must be stored in tuple space (causing tuple contention and thus latency), and processors must synchronize to change it (causing synchronization latency)

The memory model in C-Linda is very different: associative tuple space shared storage coexists with private, address-based storage. In some sense, we are still faced with a hierarchical memory model in Linda — fast, local memory (private) and Tuple Space (shared), except that any shared data must be copied into local memory before being used, including potentially large data structures such as matrices. The programmer is helped by this clear distinction between shared storage and local storage, but mandatory copying of shared data also bothers the programmer and hinders performance, although only those parts of the shared data structures that must be used need to be copied. Matrices, images, *etc.*must be cut up into chunks when placed into tuples, else access to the whole structure will be serialized if placed in a single tuple. This is a data partitioning problem: what is the appropriate number of chunks? If the data partitioning too coarse, each tuple holds too large a part of a data structure. Excessive serialization results because of contention for a single tuple. If the data partitioning too fine, there is more overhead because of more frequent access to TS.

For example, partitioning an iconic image data into tuple chunks is not obvious. Should the tuples be made according to spatial distribution (*e.g.* one row of the image per tuple)? Or should

they be according to image properties (*e.g.* one tuple per region)? Of course, this is dependent on the algorithm, but also on the Tuple Space implementation. Additionally, a single tuple may sometimes be more convenient for programming, as there are fewer `in()` and `out()` operations to perform. The advantage in C-Linda is that because of the semantics of TS, operations on shared memory (TS) are atomic.

Synchronization in Linda is explicit, but is much simplified, because of the uncoupling and anonymity provided by associative tuple matching, which removes the naming problem between communicating processes, and the atomicity of tuple space operations However, because synchronization is still explicit, it is still easy to create a non-deterministic program by forgetting synchronization statements. The programmer must still determine where to serialize execution (mutual exclusion), although the mechanism to do so is quite simple.

Explicit synchronization in C-Linda is problematic in other respects. For example, programmer must often ask whether an `in()` or a `rd()` is the appropriate operation, or, in other words, whether the data they are dealing with is read-only or not. If a task modifies the data it obtains from tuple space, an `in()` is necessary, to remove the old data from shared memory (tuple space). Doing a `rd()` instead of an `in()`, followed by an `out()`, will leave the old data in tuple space, and place the new data in tuple space as well. Tasks retrieve tuples of identical size and type non-deterministically, so that this unintended sequence of operations would lead to a non-deterministic program

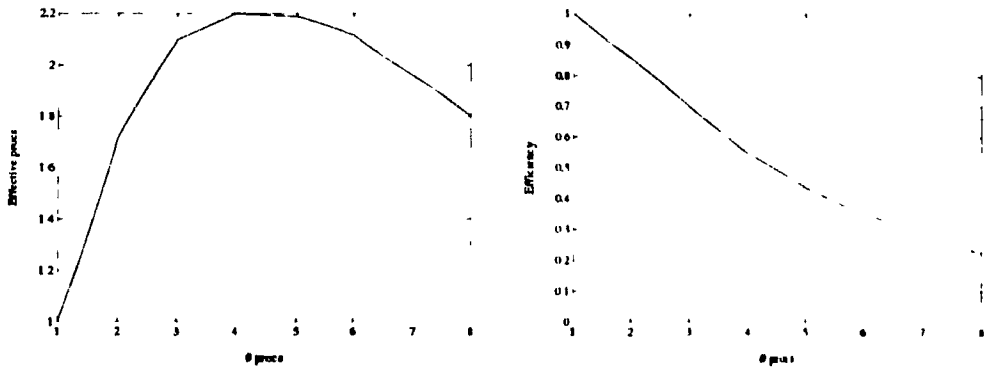
The very nature of tuple space is a problem for program modularity For example, if two copies of a function read from tuple space, we must synchronize their accesses to tuple space so that one invocation does not access the tuples that were destined to the other Similary, two copies of a function writing to tuple space must place a marker in their output tuples to distinguish them These problems arise because Linda TS is flat [20]: any module can interact with any other module, anonymously, through tuple space, which is obviously an impediment to program modularity. Tuple space's great strength for *synchronization* is a great handicap when storing data in shared memory, as any module can side effect another. Modular program construction would require that one invocation of a function not interfere with another; however, in the example above, this would be the case, as the Linda model is based on side effects (to Tuple Space), and tuple space is flat, not partitioned, to allow for arbitrary synchronization. The above examples show how easy it is to forget synchronization statements that will make a program non-deterministic The way to synchronize

between simultaneous invocations of the same function is usually to have a single counter tuple in tuple space, read by tasks and atomically incremented to provide a unique identification stamp for output tuples placed in tuple space. This way, each invocation of a function can uniquely stamp the tuples it produces.

A minor programmability distraction is caused by the nature of the associative access to tuple space Conventional memory is accessed through its address (pointers), Tuple Space memory through its contents (values), so that when mixing the two in the form of C-Linda, some duplication of names for a single entity is unfortunately needed. A common way of doing this is by inserting a string identifier in a data tuple to be able to associatively match to this tuple in tuple space. This string identifier becomes the "name" of the data structure. For example, a typical tuple might be `("vector a", 2, 15.0)`, where `"vector a"` is the name of the overall data structure, whereas when it is read into a process' memory, vector a's name becomes `float *a`. We now have two names for vector a depending on where it is stored, `float *a` in a process' local memory, and a character string, `"vector a"`, in tuple space In fact, identifiers are the only way to pass tuple space data structures as arguments to functions: they are used as a "handle" to the data structure in tuple space

3.5. Sequent Balance C-Linda Experimental Results

The experimental procedure carried out was identical to that described in section 3.2 However, the nature of the C-Linda implementation on the Sequent Balance is quite different from that on the U S. on the BBN Butterfly. In C-Linda on the Balance, each task (`eval()`) is implemented as a UNIX fork operation; if a processor is free on the machine, the forked task will be scheduled there. If not, the task will be scheduled on an already-busy processor, and context switching will result. Additionally, there is no way for a user to restrict the number of processors available for use, for testing purposes Instead, the programmer must restrict the number of tasks created. This coupling of number of processors and task granularities is undesirable for testing purposes. Such a scheme is simply *static load balancing*, or the adjustment of task size to match processor numbers to obtain high utilization, rather than *dynamic load balancing*, where the number of tasks is unrelated to the number of processors, and high utilization is obtained by having short tasks such that task starvation



(a) Speedup (b) Utilization

Figure 3.6: Speedup and efficiency for C-Linda on the Sequent Balance

effects of $p-1$ processors waiting for a single processor to finish a longer task are minimized. In terms of intermediate-level vision processing, dynamic load balancing is far preferable, as the size of tasks generated by the algorithm will vary according to the data sets. With C-Linda on the Sequent Balance, dynamic load balancing requires too fine a task size to be practical: the UNIX fork call is expensive because it involves making a duplicate of the calling process' address space. Additionally, creating more tasks than there are available processors will simply produce UNIX process-level context switching, under Sequent Balance C-Linda, which is a source of undesirable overhead. Therefore, creating small tasks is not possible, and static load balancing was the preferred choice to avoid overhead.

Speedup and utilization results are shown in figure 3.6. Note that now, because of static load balancing, granularity and processor numbers are coupled: a particular granularity corresponds to a particular number of processors. Tests were therefore performed for summation granularities of 6200, 3100, 2075, 1550, 1250, 1050, and 775, which correspond to 1, 2, 3, 4, 5, 6, and 8 processors, respectively, for a 6173-point data set, as the reader can verify.

The poor speedup and utilization curves simply reflect the coarse granularity supported by this implementation of C-Linda on the Sequent Balance, not of the C-Linda model itself, which can be as fine-grained as the programmer chooses to make it. However, these results do indicate that dynamic

task creation on the Sequent Balance should be carefully examined by the programmer and compared to the minimal task size supported by the architecture. Once more, the programmer might have to restructure his or her code to suit the hardware.

3.6. Comments on Id Programming

Id on the MIT TTDA is a very powerful programming tool for intermediate-level vision programmers in particular, and scientific parallel programmers in general. Its strengths are quite clear. First, Id provides *logical independence*. The programmer is freed from the details of parallelism, as Id parallel programming is implicit. Second, Id provides *closeness to problem domain*. Through the use of higher-order functions and declarative programming, an Id program is close to the algorithmic specifications for the problem. A third strength is *fine grained parallelism*: Id imposes no artificial sequentiality onto a program. All the parallelism present will be exposed, down to the instruction level. Finally, *determinacy* is one of the most important strengths. Id programs are guaranteed to be determinate, producing the same results regardless of run-time configuration.

However, Id on a dataflow architecture is no panacea. There are indeed several areas where a programmer might have some difficulty. First, Id does not provide any way of controlling operational behavior in cases where it would be desirable to do so. There is thus *absence of control over operational behavior*. Second, it is easy to code an algorithm that generates enough parallelism to *overwhelm finite machine resources*. Next, there can be *excessive dependence on compiler*. A functional language does not remove the necessity for the user to provide adequate algorithms and data structures, although it is tempting for the user to forget. Finally, there is the *rarity of real implementations*. Id is implemented on very few machines, which are not presently generally available.

Let us once again return to our evaluation criteria.

Arguments for the closeness to problem domain of functional languages, including Id, have been made elsewhere (see appendix B), and will not be repeated. Functional language programs can easily be considered as executable specifications, and are closer to mathematical descriptions of

scientific problems than any imperative language description because of their complete reliance on expressions. This naturally applies to intermediate-level vision algorithms as well

Id is well-suited to scientific applications in general. In terms of intermediate-level vision programming, because there is no notion of partitioning, nor distance in communication, any low-level and intermediate-level vision algorithm can easily be implemented The architecture we are examining to support these programs is a general-purpose dataflow machine, the TTDA It supports very fine-grained parallelism, which could be used to exploit the fine-grained parallelism potentially present in intermediate-level vision algorithms. The effectiveness of Id on other architectures would obviously vary depending on the architecture, the compiler, and the code itself.

The task creation mechanism in Id is beyond programmer control, and therefore determined by the compiler. In some sense, this lack of mechanism provides the most convenient task creation mechanism of all. In the case of the TTDA dataflow architecture, each instruction is a task

Paradoxically, this source of great strength for the Id programming approach can also be a source of great problems. In a dataflow architecture, partitioning a code into threads of sequential execution has no meaning. However, on a more conventional architecture, such issues, as well as scheduling issues, would probably have a great deal of importance. On such architectures, Id provides no way of defining operational behavior if an optimal parallel algorithm is known, or refining its behavior if performance requirements demand it. Para-functional programming [25] provides *annotations* that allow for finer control over operational behavior, without completely restructuring the program, while keeping its functional nature. Even if such annotations are provided, Hudak [25] believes there still will be cases where complete restructuring of code for performance will be necessary Indeed, it is shown in [35] that optimal compile-time scheduling is an NP-complete problem, although efficient approximations can be found. The Id compiler used for this thesis is tailored to extract parallelism on a particular architecture — although this portability question may be less significant if it can be shown that the dataflow architecture is the proper one for scientific parallel processing

These questions are all related to the issues of expressiveness, efficiency, and parallelism, described by some as a tri-polar relationship of mutual repulsion [16] For example, if all parallelism is explicitly written in the program, it will become cumbersome (an example of inexpressiveness). Other examples include memory usage, higher-order functions, and dealing with finite machine

resources

memory usage: a central issue is the extravagance in memory usage that results from the conceptually clean functional model that deals with values rather than storage cells; imperative languages derive efficiency from being able to optimize the storage and movement of data.

higher-order functions: at the current level of compilation technology, these abstractions result in decreased efficiency, depending, of course, on the architecture chosen.

finite machine resources: Id brings expressiveness and parallelism together without degrading either; the finest level of parallelism is exposed, such that all possible operations are as asynchronous as possible. This approach requires a *run-time mechanism to check and defer arbitrary-sized activities until their preconditions are satisfied* (*e.g.* a dataflow architecture). For example, using recursion to set up parallel function execution can generate a lot of potential parallelism. However, in most architectures, a straightforward implementation that spawns each function call as a task will incur large overhead that will offset much of the gain that parallel execution brings. Simply looking at speedup curves might be misleading, as it may lead the user to assume that he/she has found a good algorithm when in fact a serial algorithm would take less time. The dataflow challenge is to manage this explosion of activities under finite resources. The more conventional solution is to have sequential processors execute short sequences of operations. This approach raises the questions of how efficiently sequential architectures can switch short tasks, and of the effectiveness of compilers in partitioning to create the appropriate task grains.

If functional languages need unconventional apparatus to achieve their efficiency (*e.g.* dataflow), their use will be curtailed. Thus in the tri-polar relationship discussed above, in many cases some compromise must be made in expressiveness and parallelism in order to demonstrate a decisive advantage in efficiency.

Id parallel programming is entirely implicit. There are no parallel annotations, no partitioning, scheduling, synchronization, nor memory latency to worry about. The latter two considerations are solved by the TTDA architecture. Synchronization is provided at the finest grain possible [7]. I-structures are provided to exploit the highest degree of parallelism present, through fine-grained

synchronization causing premature reads to be queued until the data arrives. Operands to an instruction in the dataflow architecture must all be present for the instruction to fire, thus creating synchronization at the instruction level. Task switching occurs at the instruction level; as all memory reads are split-phase, the memory latency problem is solved by executing an arbitrary number of instructions until the memory request is completed, irrespective of whether the wait for data is because of contention for a particular memory or is simply because of the remoteness of the memory.

The issues of lightweight task creation and easy load balancing are also solved by the architecture. The finest grain parallelism possible is exposed by the functional nature of Id, and tasks are equivalent to instructions on the TTDA Load balancing among processors of a multi-processor TTDA machine is done by distributing copies of the code to each processor, and applying a hash function on token tags to determine which processor will handle the execution [7]

Memory latency problems are solved in the architecture, which makes handling large, shared data structures transparent to the programmer, and efficient on execution. In terms of the TTDA architecture, all memories are globally addressable Good performance therefore does not depend on a programmer carefully mapping the data to memory and being familiar with the storage model of the architecture.

Determinacy is another strong point of the system, and is guaranteed by Id's functional nature. The importance of this fact has been discussed extensively elsewhere in this thesis, and will not be discussed further here.

Communication between tasks does not have much meaning in the domain of implicit parallelism, being out of the programmer's control. In the TTDA, the production of results needed by other tasks (*i.e.* instructions) produces synchronization at the instruction level

3.7. Potential Problems with Id Approach

We have already mentioned the problems with the lack of control over operational behavior. Annotations to specify partitioning and scheduling would be useful to programmers who know optimal algorithms or want to refine the performance of existing code on many conventional MIMD architectures, although the relevance of these on a dataflow machine is unclear. However, the importance of being able to control excess parallelism is clear. In fact, the TTDA provides, at the architecture level, a mechanism for throttling the concurrency of loop iterations, allowing only k iterations at once [7]. Throttling of function invocations is not performed, so overloading of machine resources is therefore still possible

Excessive programmer dependence on the functional language compiler is also a subtle, but real, problem. If the reader refers to the sample functional program in section 2.3.3, he or she will notice that the functions `row` and `col` do not perform any useful work — they only produce copies of a matrix' rows and columns, which is quite inefficient It is very tempting for a programmer to code the matrix multiplication algorithm in that way, however In fact, programming details can be put into five categories[16]

1. *Algorithmic details:* details describing a method of the solution.
2. *Data structures:* details specifying an organization of the program's data.
3. *Control details:* details defining an order of the program's operations.
4. *Type related details.* details specifying types of the program's variables.
5. *Storage related details:* details describing representations of the program's variables.
6. *Resource related details.* details that specify changes in allocation of a program's resources.

Functional languages with implicit parallelism remove all details except the first two. The responsibility for adequate algorithms and data structures is in no way lessened by the use of a functional language. Here is a more efficient version of matrix multiplication [28] than the program shown in section 2.3.3. The `matmult` function is kept intact; only the support functions are changed.

```
def row i X k = X[i, k];

def col j X k = X[k, j];

def ip rowC colD = {
     (_,n) = bounds rowC;

   in
     sum (1,n) {fun i = (rowC i)*(colD i)}
   };
```

In this way, `row` and `col` do not copy a row or a column. `matmult` invokes `row i A` and `col j B`, which causes partial applications of `row` and `col` to be made, creating two new functions of a single argument each. In turn, these new functions are passed to `ip`, the inner product function, which invokes them as `rowC i` and `colD i`, with the single remaining argument being the index `i`. In this way, higher-order functions (those created by the partial application of `row` and `col`, and passed as arguments to `ip`) remove the need for copying and increase efficiency. A programmer must be able to take advantage of these possibilities.

The final disadvantage of Id is the rarity of implementations on real machines, which dramatically curtails is usefulness as an intermediate-level vision research tool. Indeed, the experimental code used to demonstrate Id's usefulness has been implemented on a *software simulation* of the TTDA, called GITA, the Graph Interpreter for the Tagged-token Architecture, described earlier in this chapter. Unfortunately, this is of limited use as a tool for intermediate-level vision research, because of the obvious slowness of a software simulation of a hardware architecture.

These arguments may be made obsolete by the commercial introduction of the Monsoon dataflow architecture, developed at MIT [15]. Monsoon is a significant modification of the TTDA to remove the associative matching of input tokens to an instruction. A single processor prototype has been operational at MIT since October 1988, and Motorola Corporation is collaborating with MIT in building multiprocessor versions. These machines are all programmable in Id

3.8. Id Experimental Results

Ironically, the flexibility and heavy instrumentation of the Id World TTDA architecture simulation, combined with the very nature of the experimental problem, prevents us from presenting complete measurements of cooperative fitting on the TTDA, except for extremely small data sets ($\leq$ 10 data points). This is because of the enormous memory requirements of any simulation on the TTDA, combined with the explosion of parallelism present in the fitting problem itself.

However, what we will show is a part of the fitting process typical of the rest of the algorithm: the computation of the χ^2 merit function. This is simply a summation over all data points of (a function of) a chosen error metric. This type of summation is repeated throughout the fitting procedure, with the summand function being various functions of the error metric, or of its derivatives. This therefore exhibits a considerable degree of parallelism.

We measured various system parameters for different machine configurations. Id World allows the user to set the simulation to

idealized mode: an infinite number of processors are available, and the time taken for the result of one instruction to reach the next instruction (communication latency) is zero. This mode is useful to obtain the *parallelism profile* of the program, the maximal parallelism obtainable at each time step. In a parallelism profile in idealized mode, the number of instructions executed at each time step is constrained only by data dependencies.

finite processor mode: in this mode, both the number of processors and communication latency are adjustable. Communication latency can be ≥ 0.

The experiments we ran were the following. 1521 points were sampled on an artificially-generated ellipsoid, with added gaussian noise. The χ^2 merit function was then applied, with the error metric described in section 3.1. If p is the number of processors and l is the communication latency, measurements were taken for the following machine configurations: $p = \infty, l = 0$, $p = 30, l = 0, 1, 5, 10$, $p = 50, l = 0, 1, 5, 10$, $p = 100, l = 0, 1, 5, 10$, and $p = 200, l = 0, 1, 5, 10$. The first measurement corresponds to idealized mode; thereafter, increasing processor numbers demonstrate

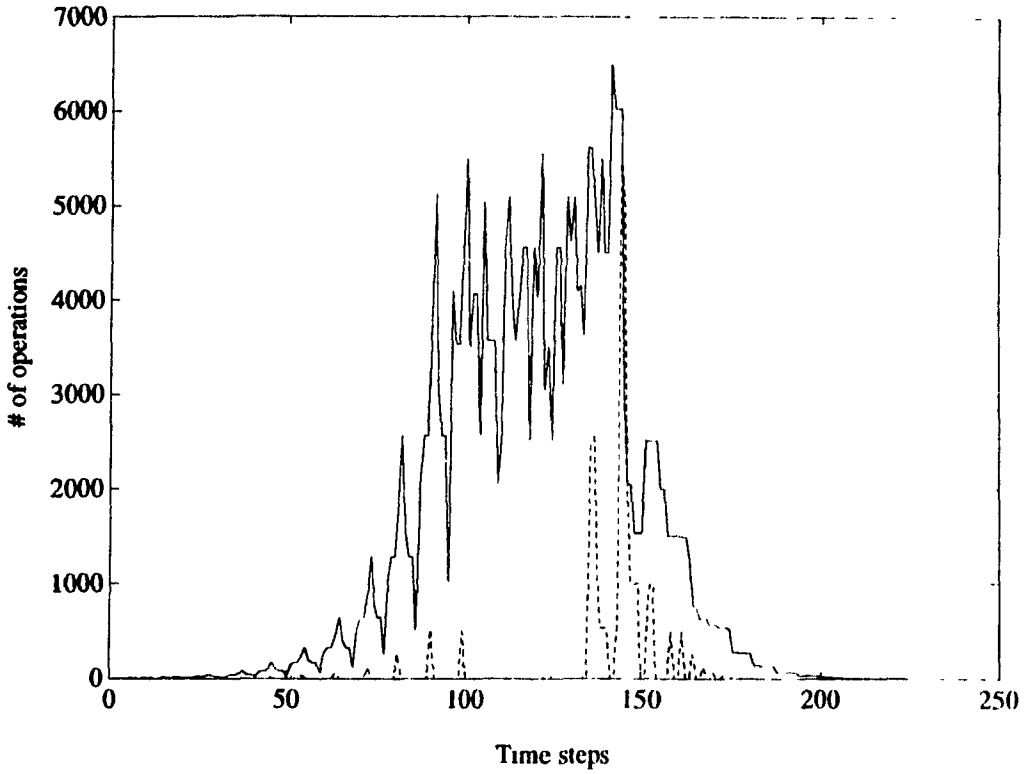


Figure 3.7: Parallelism profile for χ^2 merit function, for 1521-point surface data The solid line is the number of ALU operations, the dashed line is the number of floating-point operations, and the dotted line is the number of function invocations, at each time step

the scalability of the results, while increasing communication latency shows how performance is affected by latency.

We first show the extended parallelism profile for the program, in figure 3 7.

The parallelism profile corresponds, in the TTDA, to the number of ALU operations performed at each time step. The number of floating-point instructions is also quite interesting The summation function employed recursively decomposes the summation range into two halves, to exploit the maximal amount of parallelism available through summation. This creates an exponentially-rising number of new functions (as shown by the function invocation profile), and a large number of

floating-point operations once the leaves of the summation "tree" have been reached, which forces evaluation of the summand function (seen in the function invocation profile), and soon thereafter the large number of floating point operations.

Notice how many new functions are created by the recursive summation function — thousands of new function calls This could conceivably swamp the resources of a real machine. In such a case, the recursive summation function would have to be recoded to force evaluation when the summation range is less than some epsilon (ϵ 4, for example). This would corresponds to changing the granularity of the program to suit the architecture.

Figures 3.8, 3.9, 3.10, and 3.11 show parallelism profiles for p=30, 50, 100, 200 under different latencies. Note how little execution time changes when latency changes from 0 to 10. For example, for $p = 30$, execution time changes from 10293 to 11394 time steps, a 10.7% increase, when latency increases from 0 to 10. This is quite small, considering the increase in communication latency, and occurs because excess parallelism masks latency. Because of split-phase transactions, the dataflow processors are free to work on other instructions instead of waiting for other results. Of course, it stands to reason that when less excess parallelism is present, latency is not as well masked: for example, for $p = 200$, execution time changes from 1628 time steps to 3206, a 96.9% increase, when latency increases from 0 to 10. This is still an impressive result. increasing latency from 0 to 10 produces only a doubling of run time.

Speedup and efficiency (or utilization) curves are shown in figure 3.12. Naturally, increasing latency diminishes the speedup (or the effective number of processors), and correspondingly decreases the average efficiency, or utilization of each processor. If T_1 is the time taken by one processor (equivalently, the total number of instructions executed), and $t(p, l)$ the time taken for a machine configuration with p processors and l communication latency, then [5]

$$\text{speedup}(p, l) = \frac{T_1}{t(p, l)}$$

and

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

As can be seen from the figure, latency is responsible for flattening the speedup curves and for dimin-

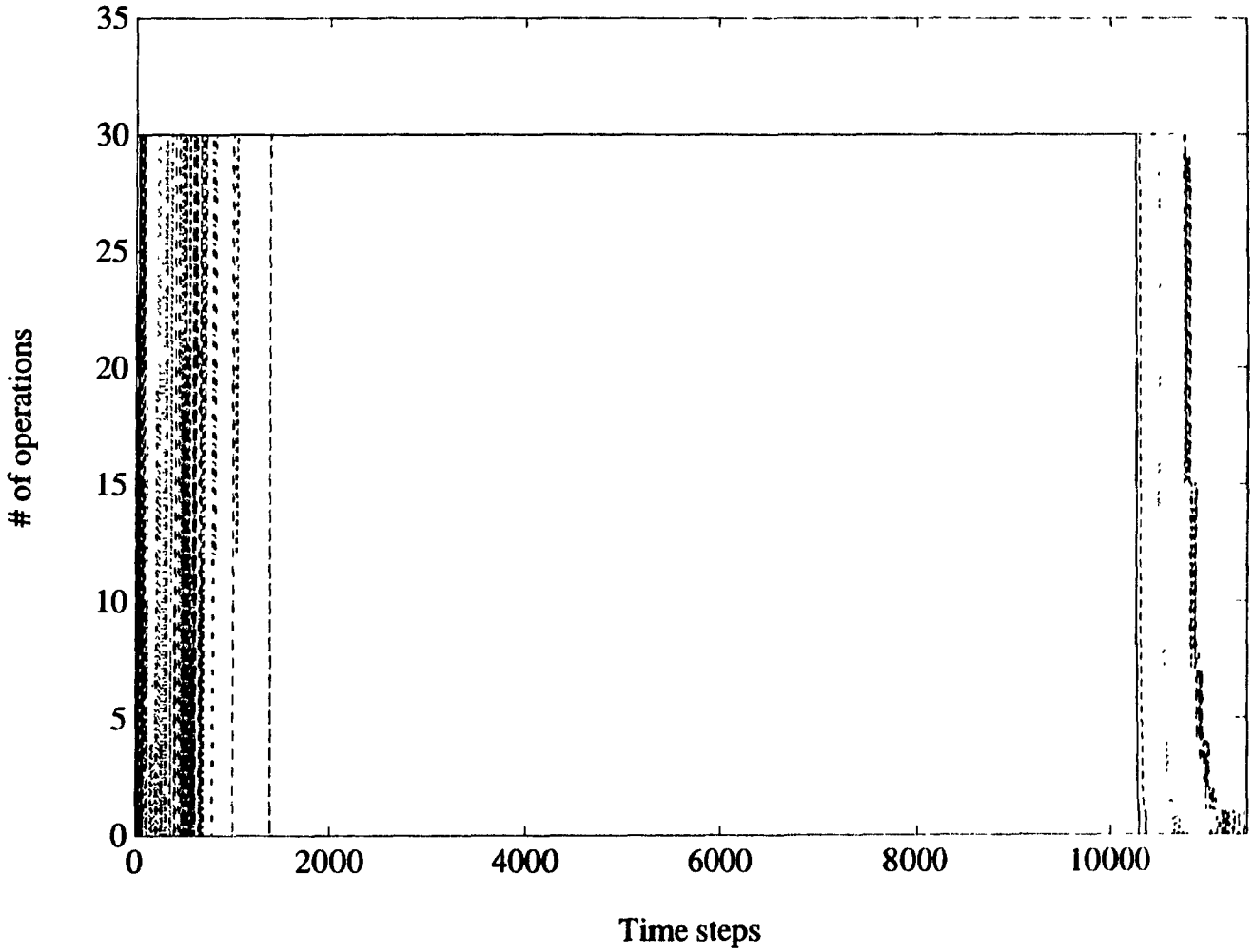


Figure 3.8: Execution time for $\backslash^2$ for $p = 30$, under latencies of $l = 0, 1, 5, 10$ (in solid, dashed, dotted, and dash-dot lines, respectively). The maximum number of operations (30, as $p = 30$) is sustainable for most of the execution time, for all latencies (from approx. $t = 1400$ to $t = 10200$). Initially, however ($t < 1400$), there is not enough work to keep all 30 processors busy, the number of operations per time step oscillates between maximum and a lower value. Obviously, the run for $l = 0$ finishes first, at $t = 10293$, then $l = 1$, $l = 5$, and finally $l = 10$ at $t = 11394$, but notice how little execution time changes as latency is increased, because excess parallelism in the algorithm is used to mask latency.

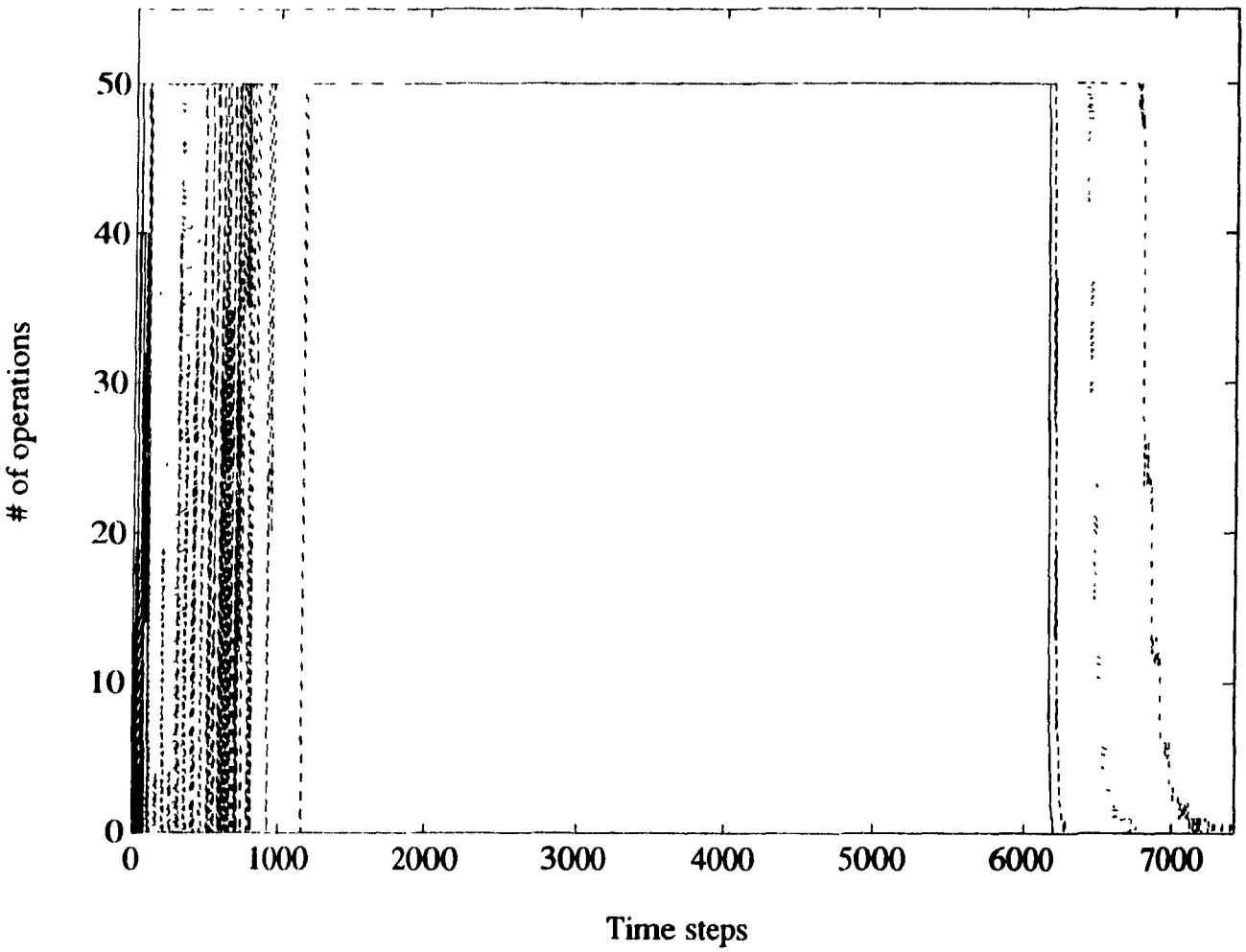


Figure 3.9: Execution time for $\backslash^2$ for $p = 50$, under latencies of $l = 0, 1, 5, 10$ (in solid, dashed, dotted, and dash-dot lines, respectively). See comments for figure 3.8. Again, notice how little execution time changes as latency is increased, because excess parallelism in the algorithm is used to mask latency.

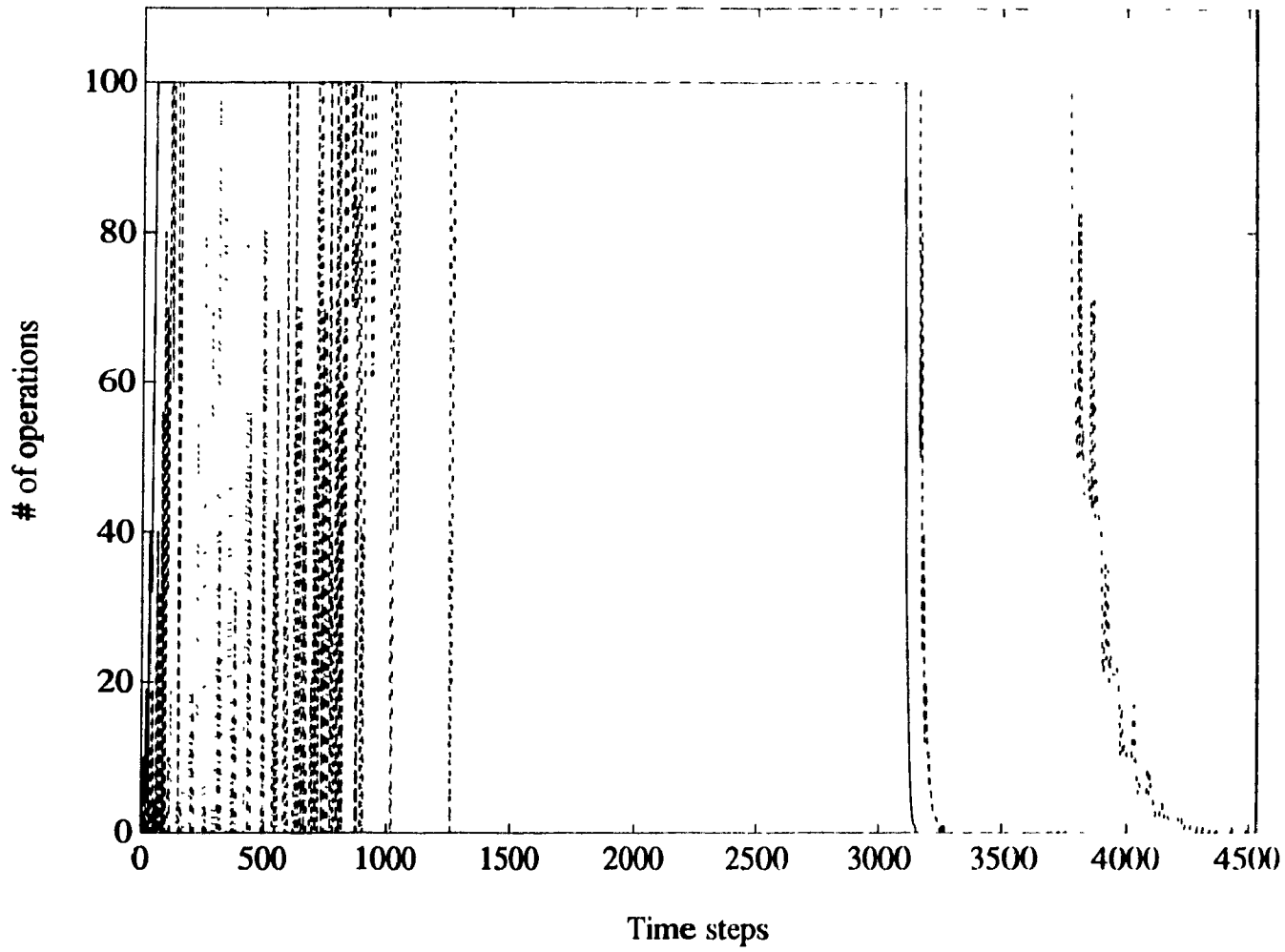


Figure 3.10: Execution time for $\backslash^2$ for $p = 100$, under latencies of $l = 0, 1, 5, 10$ (in solid, dashed, dotted, and dash-dot lines, respectively) See comments for figure 3 8. Again, notice how little execution time changes as latency is increased, because excess parallelism in the algorithm is used to mask latency.

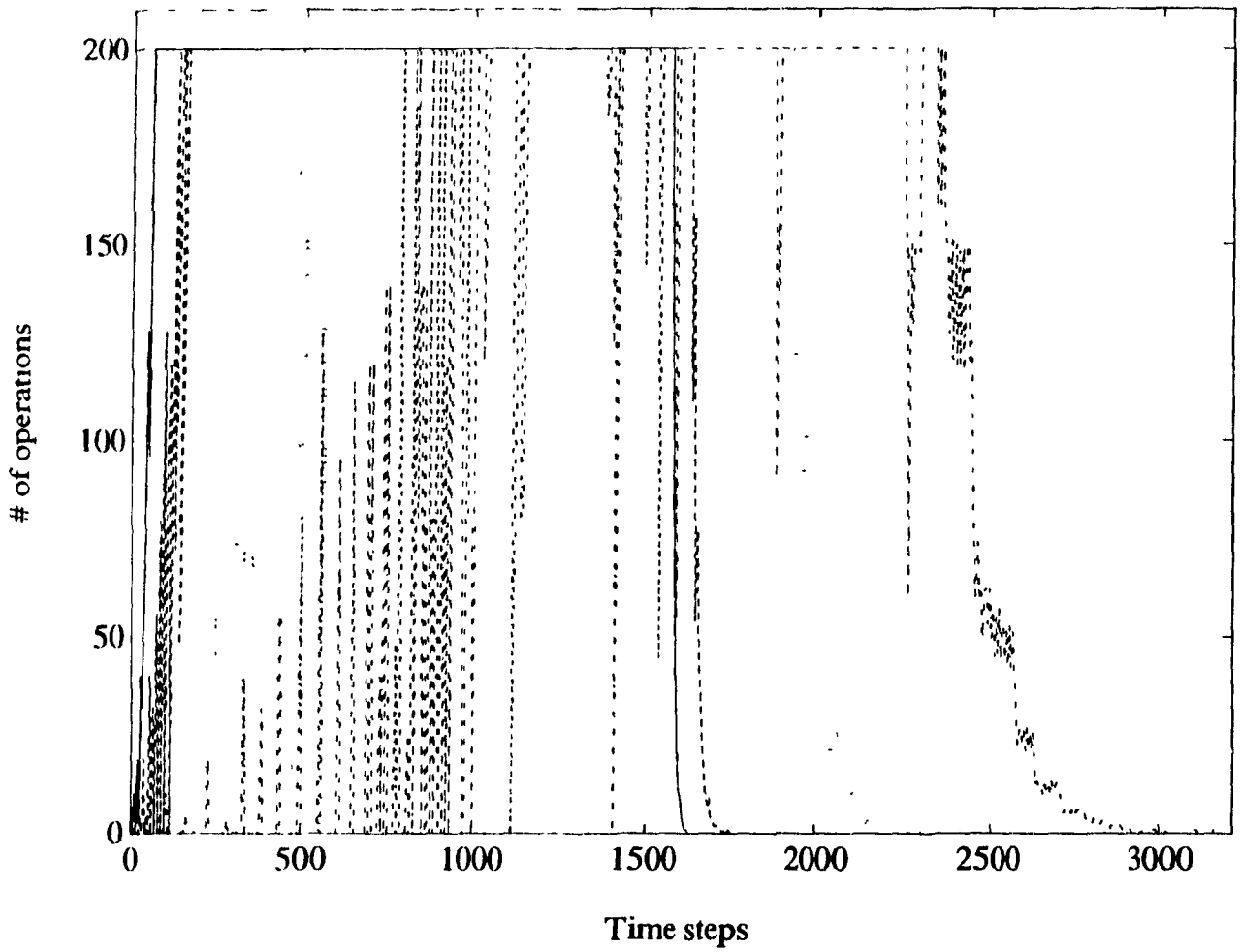


Figure 3.11: Execution time for $\backslash^2$ for $p = 200$, under latencies of $l = 0, 1, 5$ 10 (in solid, dashed, dotted, and dash-dot lines, respectively). Recall that since parallelism is used to mask latency (by keeping processors busy instead of waiting), we would expect that less excess parallelism (because of an increase in the number of available processors ($p = 200$)) would cause the effects of latency to be more apparent. Execution time increases from $t = 1628$ (for $l = 0$, solid line) to $t = 3206$ (for $l = 10$, dash-dot line), nonetheless an impressive result for such a massive increase in latency. The effects of high latency ($l = 10$) are especially apparent at startup ($t < 1000$), when little parallelism is present. This shows up as oscillations in the number of instructions executed during that time frame, as processors are busy, then idle while waiting for results, then are busy, *etc.*.

ishing average efficiency. In fact, efficiency drops to below 50% for a latency of 10, compromising the cost-effectiveness of the machine However, less severe latencies produce more acceptable results

3.9. Key Points

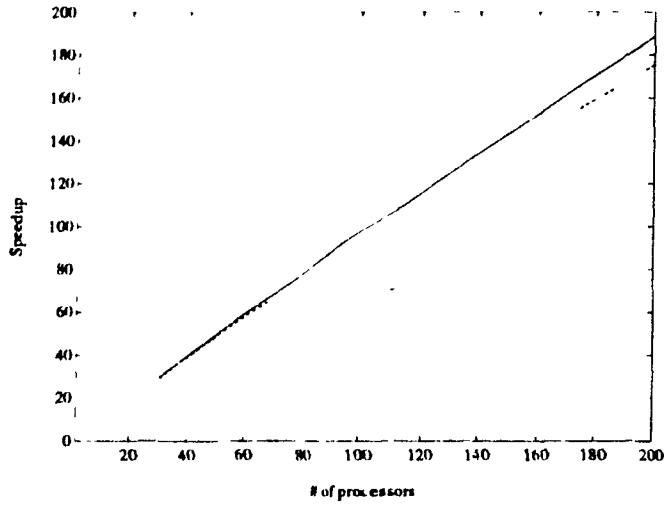
Let me summarize the key points I have made in this chapter.

Our test algorithm to compare parallel processing systems is a simplified version of parallel cooperative fitting, where different iterative fitting processes exchange information about an entity in order to obtain a better final fit. Parallel cooperative fitting is an interesting test case, as it is an iconic to aggregate transformation with large input data sets, data-dependent partitioning, and potentially high parallelism.

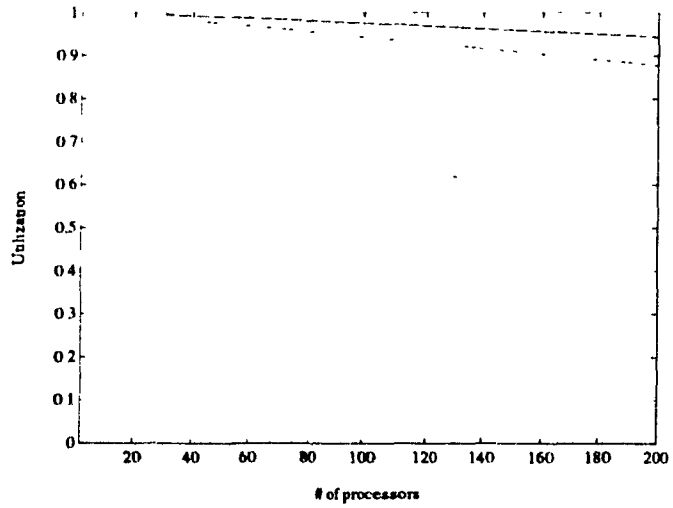
BBN US strengths are a comparatively small grain size, dynamic load balancing, large bandwidth to shared memory, and a single address space for shared data. Weaknesses are a lack of higher-order functions, an inexpressive parallelism mechanism, trial and error explicit task partitioning, and local/remote, private/shared considerations for every data object.

C-Linda strengths are uncoupled, atomic operations, flexible task creation, and load balancing. Weaknesses are a lack of higher-order functions, trial and error explicit task partitioning, explicit data partitioning, loss of program modularity, and mandatory, explicit copying of shared data

Id strengths are logical independence, closeness to the intermediate-level vision problem domain, fine-grained parallelism, and determinacy Its weaknesses are the absence of control over operational behavior, the possibilities it gives to overwhelm finite machine resources, the increased compiler reliance some users might feel, and the rarity of implementations on real hardware



(a) Speedup



(b) Utilization

Figure 3.12: Speedup and efficiency for the TTDA, in the presence of latency.

4. LESSONS TO BE DRAWN

4.1. A Comparison of Three Parallel Processing Systems

In this section we will compare and contrast our three test systems, against each other but also against our evaluation criteria.

A criticism of functional languages such as Id is excessive use of storage because of copying However, reusing storage is often possible only because of sequential execution For example, as for functional languages, the U.S. uses extra storage to allow for more parallelism. the `Share()` mechanism (and other local copies to all processors) ensure that all processors can proceed in parallel and don't have to wait for a serialized access to a single memory module.

C-Linda on the Sequent Balance, in some respects, is a better-adapted system than the BBN Uniform System for parallel programming of intermediate-level vision algorithms, but less so in others. It does not suffer from the syntactic crudeness a parallel library-based design, nor does it suffer from the semantic crudeness of a single, dominant parallelism model (of parallelism over data structures). The U.S. process generation mechanism is very inflexible with respect to the C-Linda model. With C-Linda's `eval()`, a task can be created out of any arbitrary function

The presence of what amounts to a memory hierarchy in C-Linda creates inconsistency problems, just as in the Uniform System. Modifications to shared memory may be atomic, but this does not prevent another process from using an old copy of data it obtained using a `rd()` operation. This is

an example of the synchronization problem synchronization of multiple copies of a single piece of data

The data partitioning problem is entirely analogous to that in the BBN Butterfly US: both C-Linda and the US force the programmer to explicitly cut up data structures in order to avoid serialized access. In the latter case, data partitioning was determined both by the memory model and by the C language programming model (*e.g.* C vectors on a single memory) The difference is that US data structures don't necessarily have to be copied into local memory to be used (although for performance reasons a programmer may want to), but C-Linda data structures always have to be explicitly `in()`'ed or `rd()`'ed first

Tuple space is an excellent medium for synchronization, because it uncouples the synchronizing processes and ensures atomicity As a means of inter-process communication, it keeps these attributes, but *because of the mechanisms* (`in()`, `out()`, `rd()`), the programmer is forced to make explicit copies of data, which is a burden on performance. The BBN US model is sometimes more convenient and more efficient, but always more dangerous: local and shared (remote) memory are accessed in exactly the same way This can even be done without cutting up the shared data structures into chunks, nor doing explicit copies, if performance is not affected. For example, a US scattered matrix is a lot more convenient to use for a task than a Linda matrix stored in tuple space. The latter must be `in()`'ed explicitly by any task that wants to use it. These hindrances, caused by the need to maintain atomicity in the face of changeable storage, are made irrelevant in a single-assignment system such as Id

As mentioned above, one advantage of C-Linda is its clear distinction between shared data and non-shared data. In the U.S., any pointer can point to shared memory or local memory, and the syntactic mechanisms to use a pointer to shared memory and a pointer to local memory are exactly the same: it is up to the programmer to remember which area of memory is pointed to, for correctness and performance reasons. This is extremely tedious in practice. In contrast, the distinction between shared and non-shared data in C-Linda is quite explicit, simply because of the fact that the mechanisms used for shared memory access are explicit and unavoidable. `in()`'s and `rd()`'s for shared array accesses are separate from the pointer indirection and dereferencing used in local array accesses, for example.

Let us conclude by comparing all three systems with the evaluation criteria we propose

In terms of closeness to problem domain, Id is most appropriate, as functional languages deal best with functions, the core of mathematical programming. Intermediate-level vision programming is no exception

General applicability is a feature of all three systems, which can be used for any intermediate-level vision task, but of course with varying degrees of effort in adapting and restructuring code to obtain good efficiency

Id is the most flexible system in terms of task creation: tasks are implicit (each instruction is a task) and are not a programmer concern. For explicitly-partitioned languages, C-Linda's `eval()` is more flexible than the U.S. task generators. the parallel library design of the latter means syntactic and semantic crudeness Flexible task creation is a desirable characteristic for a problem domain such as intermediate-level vision, where partitioning is data-dependent and where tasks will often be dynamically created

The overhead in task creation is the most architecture-dependent of all criteria However, the programming model also has a part to play. For example, Id imposes no artificial restrictions on parallelism; machine resources and data dependencies are the only constraints This allows for maximal parallelism on a very fine-grained machine, such as the TTDA architecture, which makes 1 task per instruction possible The BBN Butterfly supports fairly lightweight tasks, as compared to the Sequent Balance, which again is a desirable characteristic for a problem domain such as intermediate-level vision, where fine-grained parallelism will often be present However, lack of lightweight task creation is not a restriction of C-Linda, but rather of the implementation on the Sequent Balance architecture.

Determinacy is only guaranteed by programming in Id. Both U.S. and C-Linda programming rely on the programmer for program correctness, which in some case can be extremely difficult to achieve.

Load balancing is again very architecture dependent. For intermediate-level vision, where large

run time variations are possible, because of the non-uniform nature of the processing involved, load balancing is important to achieve good performance. Load balancing on the TTDA is achieved by having a task size equal to an instruction, and by the execution of code blocks across processors. On the Butterfly, fine granularity allows for dynamic load balancing, given enough tasks. On the Sequent Balance, load balancing must be static. Given the data-dependent nature of intermediate-level vision task sizes, this could be a performance problem.

Intertask communication will obviously be easiest in the presence of implicit partitioning: implicit tasks produce implicit communication and synchronization, which in the TTDA architecture are supported by the dataflow execution model and I-structures. C-Linda supports uncoupled synchronization and communication, but these are still explicit, and thus tedious to incorporate into code and error prone, as omissions will often produce non-deterministic behavior.

Handling of large, shared data structures is again easiest when out of programmer control. With Id, explicit data movement for efficiency is not needed, and the TTDA architecture solves the memory latency problem with split-phase memory transactions. With the U.S on the BBN Butterfly, high bandwidth to memory provided by the butterfly network is an excellent characteristic, but having the programmer explicitly distribute data to take advantage of this is tedious, as allocators and access mechanisms must be built for each new data structure. The programmer must also be aware that if a part of a data structure on one memory node is accessed more frequently, contention will result In C-Linda, if data is shared, it *must* be copied to local memory before being used, which is not necessary with the BBN U S. For large data structures, copying to and from tuple space may become a performance problem As for the U.S., data partitioning is also a C-Linda problem. Parallelism can be increased by decreasing tuple size, but this will increase data access overhead. For intermediate-level vision, where processing is data-dependent, data partitioning will not be easy.

4.2. Key Points

Of the three systems we examined, the best suited to intermediate-level vision programming was Id on the TTDA, because of its mathematical flavor, general applicability, lightweight task creation,

determinacy, and the logical independence it provides the programmer. U S. programming on the BBN Butterfly and C-Linda programming on the Sequent Balance do not have the mathematical expressiveness of functional programming, and often force the programmer to restructure her or his code through trial and error explicit partitioning

The BBN Uniform System also imposes the burden of managing the system's hierarchical memory on the programmer, has an inexpressive, library-based parallelism model, and does not guarantee determinacy, but supports fairly lightweight task creation and simplifies programming through a single, shared memory space.

C-Linda creates a data partitioning problem in tuple space for the user, does not guarantee determinacy, and the Sequent Balance can only handle very coarse-grained tasks However, C-Linda parallel programming, while explicit, is uncoupled and atomic

5. CONCLUSIONS

The problem investigated by this thesis was that of giving intermediate-level vision researchers adequate parallel processing tools for their work, where data and computational structures do not fit the SIMD execution model, but require a MIMD execution model instead.

Scientific programmers usually refine the abstract models and algorithms which their programs implement, and should not be expected to be parallel architecture experts. Therefore, their tools should be adapted to their problem domain, for fast coding, and should provide logical independence from solving the four crucial MIMD issues of parallel processing, partitioning, scheduling, synchronization, and memory latency Current tools, such as Uniform System programming on the Butterfly and C-Linda programming on the Balance, do not.

Functional languages, on the other had, as exemplified by the Id language running on the TTDA architecture, are a more appropriate solution. They are close to the scientific problem domain, because they are based on the function and on expressions. They are amenable to compiler solutions to the cited problems, because they do not allow a programmer to specify restrictive commands, or to artificially restrict the order of execution. This allows a compiler to extract all the parallelism present in a program.

Our demonstration about Id on the TTDA seems to rely strongly on the dataflow architecture to solve the memory latency problem (through split-phase memory transactions). But it is precisely the high parallelism available in a functional program that allows the use of this architectural feature.

More generally, it is obvious from this thesis that it is impossible at the present time to completely

decouple parallel architectures and parallel languages — in effect, impossible to provide complete logical independence However, functional language systems come closest, and ofter by far the best opportunities for parallel execution.

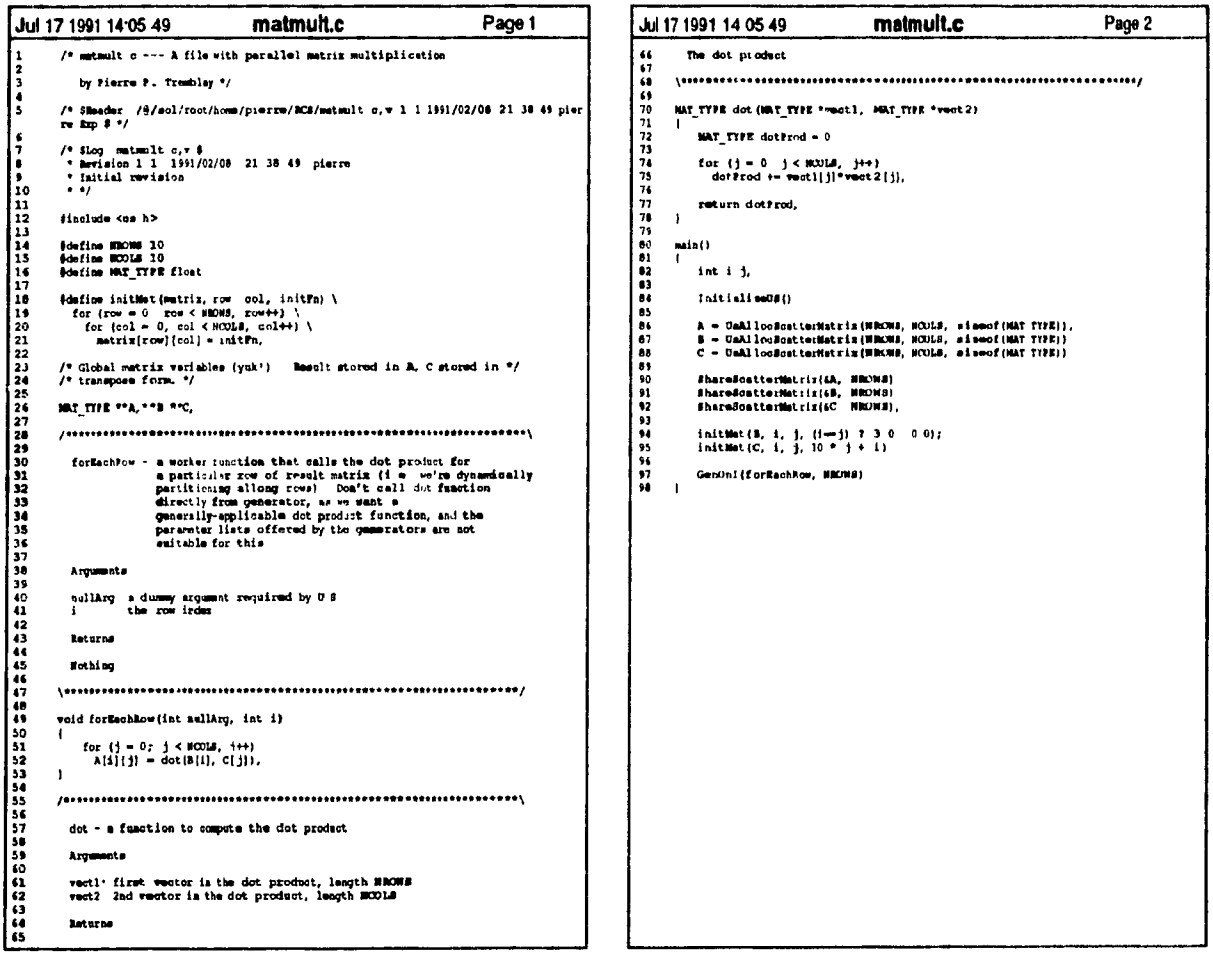
Why then aren't such systems commonly available? There are a few reasons, which we mentioned in chapter 1. Extensions to familiar sequential languages — through parallel constructs or high-level interfaces to libraries — are more likely to appeal to scientific programmers than are new concurrent languages. There also is an apparent ease in parallelizing sequential programs using extensions or libraries, as there is minimal rewriting. A further reason is the availability of production-level compilers for parallel machines. Most scientific programmers' programming experiences are with imperative languages such as C and Fortran, and the overwhelming majority of current scientific codes were written in Fortran. Parallel architecture vendors are therefore likely to continue work on parallelizing compilers for Fortran, instead of compilers for functional languages Additionally, functional language research is quite a young field, applying it to parallelizing scientific programs will inevitably require some time. It is hoped, however, that this thesis has demonstrated their clear advantages over imperative language systems for the scientific programmer

A. A UNIFORM SYSTEM PROGRAM EXAMPLE

We present a very simple Uniform System program for matrix multiplication in figure A 1. Notice especially:

- The memory allocation calls `UsAllocScatterMatrix()` and `ShareScatterMatrix()`. `UsAllocScatterMatrix()` allocates shared memory for a C pointer-to-pointer matrix, so that matrix rows are spread across the memory nodes of the machine. A normal C matrix would be stored in a single memory, and would be a source of contention if access by multiple processors `ShareScatterMatrix()` makes local copies of the row pointers of such a "scattered" matrix. The vector of row pointers is accessed for every element access in the matrix. This vector is kept in a single memory location, which is a source of contention if the matrix is accessed by many processors simultaneously. `ShareScatterMatrix()` makes a local copy of this vector of row pointers on each processor. These two function calls are examples of how the programmer has to keep in mind and adapt her program to the memory model of the machine, to obtain good performance

- The task generator call `GenOnI()`. In this case, the tasks to execute are the function `forEachRow()`. Each invocation is passed an argument index by `GenOnI()`, which in this case runs from 0 to NROWS. This function is an example of the somewhat restrictive task creation mechanism in the Uniform System. Each identical task is passed a single argument, an integer index in this case. Additionally, as in every system where partitioning is explicit, it is unclear whether or not the size of the generated task is appropriate for the granularity of the machine. Any change in the granularity of the tasks would involve some restructuring of the code.



Jul 17 1991 14:05 49 matmult.c Page 1

```
1     /* matmult c --- A file with parallel matrix multiplication
2
3        by Pierre P. Tremblay */
4
5     /* $Header  /@/sol/root/home/pierre/BCS/matmult c,v 1 1 1991/02/08 21 38 49 pier
      re Exp @ */
6
7     /* $Log  matmult c,v @
8      * Revision 1 1  1991/02/08  21 38 49  pierre
9      * Initial revision
10     * */
11
12    #include <os h>
13
14    @define NROWS 10
15    @define NCOLS 10
16    @define MAT_TYPE float
17
18    @define initMat(matrix, row  col, initFn) \
19      for (row = 0  row < NROWS, row++) \
20        for (col = 0, col < NCOLS, col++) \
21          matrix[row][col] = initFn,
22
23    /* Global matrix variables (yuk!)    Result stored in A, C stored in */
24    /* transpose form. */
25
26    MAT_TYPE **A,**B **C,
27
28    /*************************************************************************\
29
30      forEachRow - a worker function that calls the dot product for
31                   a particular row of result matrix (i e  we're dynamically
32                   partitioning along rows)   Don't call dot function
33                   directly from generator, as we want a
34                   generally-applicable dot product function, and the
35                   parameter lists offered by the generators are not
36                   suitable for this
37
38      Arguments
39
40      nullArg  a dummy argument required by U S
41      i        the row index
42
43      Returns
44
45      Nothing
46
47    \*************************************************************************/
48
49    void forEachRow(int nullArg, int i)
50    {
51        for (j = 0; j < NCOLS, j++)
52          A[i][j] = dot(B[i], C[j]),
53    }
54
55    /*************************************************************************\
56
57      dot - a function to compute the dot product
58
59      Arguments
60
61      vect1' first vector in the dot product, length NROWS
62      vect2  2nd vector in the dot product, length NCOLS
63
64      Returns
65
```

Jul 17 1991 14 05 49 matmult.c Page 2

```
66      The dot product
67
68    \*************************************************************************/
69
70    MAT_TYPE dot (MAT_TYPE *vect1, MAT_TYPE *vect2)
71    {
72        MAT_TYPE dotProd = 0
73
74        for (j = 0  j < NCOLS, j++)
75          dotProd += vect1[j]*vect2[j],
76
77        return dotProd,
78    }
79
80    main()
81    {
82        int i j,
83
84        InitializeUS()
85
86        A = UsAllocScatterMatrix(NROWS, NCOLS, sizeof(MAT TYPE)),
87        B = UsAllocScatterMatrix(NROWS, NCOLS, sizeof(MAT TYPE))
88        C = UsAllocScatterMatrix(NROWS, NCOLS, sizeof(MAT TYPE))
89
90        ShareScatterMatrix(&A, NROWS)
91        ShareScatterMatrix(&B, NROWS)
92        ShareScatterMatrix(&C  NROWS),
93
94        initMat(B, i, j, (i==j) ? 3 0   0 0);
95        initMat(C, i, j, 10 * j + i)
96
97        GenOnI(forEachRow, NROWS)
98    }
```

1

Figure A.1: BBN Uniform System matrix multiplication code

B. PARALLEL FUNCTIONAL PROGRAMMING

In this section we will look at parallel functional programming. The issues presented will be:

- what functional programming is,
- the parallel programming problems it solves, and how it solves them.

We will also look at an example parallel functional program and show, under idealized conditions, how much parallelism it holds.

B.1. Functional Programming

This section will give a brief summary of what functional programming is. It will not be an exhaustive look at the history of functional languages, nor will it be a thorough overview of all characteristics of functional languages: we will only touch on those characteristics that are important to computer vision programmers. The interested reader is directed to [24] for an excellent survey article on functional languages.

We start by characterizing programming languages as either *imperative* or *declarative*.

imperative languages: characterized as having an implicit state that is modified (*i e.* side effected) by constructs (*i e.* commands) in the source language

declarative languages: have no implicit state, emphasis placed entirely on programming with expressions.

Imperative languages include the most used languages today, such as Pascal and the C language In these languages, programming is split [9, p. 639] into

- an orderly world of expressions (*e.g.* `f(a+b) + c(f(d))`
- a disorderly world of statements, with few useful mathematical properties

The world of statements is represented primarily by the assignment statement [24, p. 361], whose effect is to alter the underlying implicit store (*i e.* the computer's memory) so as to yield a different binding for a particular variable. This has important consequences for parallel programming, as we shall see later.

Declarative [21, p. 305] languages allow the programmer merely to state what should hold true with respect to a computation, without bothering to say precisely how the computation should be done. Functional languages [24, p. 360] are declarative languages whose underlying model of computation is the function[1]

We briefly describe the characteristics of functional languages. Again, the reader is referred to [24, 9, 21] for more detail. Pure functional language characteristics include:

- Complete freedom from side effects: coding is merely defining expressions and functions. *Nothing is ever modified or reassigned*, as there is no assignment statement to give same variable a different binding.

[1] In contrast to the relation that forms the basis for logic programming languages.

- Functions are basic program building blocks, and can be passed around as arguments (*higher-order functions*)
- The order of evaluation of expressions is unimportant (the Church-Rosser property).

We will examine the consequence of these characteristics in later sections. The interested reader can examine a simple functional program in appendix 2.3.3

B.2. Characteristics of Modern Functional Languages

Even though they are somewhat orthogonal to parallelism, some of the features of modern functional programming languages will be discussed below, because of their importance for expressiveness. The reader is referred to Hudak [24] for a much more thorough treatment of the matter. I will follow [24] in this exposition.

As stated by Hudak, a function is an abstraction of some common behavior (*i.e.* the production of results) over values (*i.e.* the arguments). If a language allows functions to be stored in data structures, passed as arguments and returned as results, then the language is said to allow *higher-order functions*. Higher-order functions are a characteristic of functional languages that greatly enhances the expressiveness of the language. In appendix 2.3.3, we show how the behavior of summation is abstracted over any possible arguments.

Non-strict evaluation of expressions can take two forms, either *lazy evaluation*, or *eager evaluation*. Lazy evaluation is also referred to as *call-by-need*; an expression will not be evaluated unless it is needed in a computation. This frees a programmer from efficiency concerns about not evaluating an expression unless absolutely necessary; for further details, see [24].

Eager semantics imply that in a function application such as `(f x)`, the body of `f` and the evaluation of `x` *will proceed in parallel* [8]. While this will potentially increase the available parallelism obtainable in a program, there also exists the possibility of wasting resources on unnecessary com-

putation, thus the usefulness of explicitly specifying evaluation of expressions in a delayed manner, as explained above.

User defined data types, representation and implementation abstraction and hiding are another important characteristic of modern functional languages, enhancing modularity, code clarity, and facilitating debugging through better type-checking

Pattern matching allows the programmer to write several equations when defining the same function, only one of which is applicable in a given situation [24]. For example, in Id, the factorial function might be described in this way, using pattern matching.

```
def  fac 0 = 1
 |.. fac n = n*fac(n-1);
```

Note how intuitive this definition of the factorial function is.

Array comprehensions are non-strict data structures that treat the array as a single entity defined declaratively, rather than as a global object holding values, updated incrementally [24]. Array comprehensions thus specify the shape and the contents of an array simultaneously [27]. Array comprehensions are especially expressive. for example, we can express array elements constructed from recurrence relations quite easily, as the following Id code suggests [27]:

```
A = {matrix (1,n),(1,n)
     | [1,1] = 1
     | [i,1] = 1            || i <- 2 to n
     | [1,j] = 1            || j <- 2 to n
     | [i,j] = A[i-1,j]+
               A[i-1,j-1]+
               A[i,j-1]     || i <- 2 to n
                            &  j <- 2 to n};
```

The programmer is thus freed from worrying about the order in which the elements should be evaluated.

B.3. Suitability of Functional Programs for Parallel Execution

Functional programs are well-suited for parallel execution because of three (related) characteristics [29, p. 2]

- parallelism is implicit in their operational semantics The programmer does not explicitly break up a task into parallel components (no explicit partitioning), and so does not worry about synchronization.
- Functional programs are *determinate*, *i e* , the result of a functional program depends only on its inputs, and never on the machine configuration or the runtime scheduling policy. This is a *major* simplification in debugging.
- Most importantly, the *only limits on parallelism* are from *data-dependencies* and finite machine resources

These characteristics deserve some comment.

Implicit parallelism [21, pp. 338–339] in declarative languages means that declarative language programs may be executed on a parallel machine—but they don't allow programmers to state *explicitly* how parallelism is to be created and controlled, that is, how partitioning, scheduling and synchronization are to be performed [3, p. 125] Instead, in functional languages the parallelism is implicit and supported by the underlying semantics [23, p 61]. There is no need for special message-passing constructs or other communications primitives, no need for synchronization primitives, and no need for special "parallel" constructs such a "parbegin parend", all of which are needed in explicitly parallel schemes. The compiler detects the parallelism and generates calls to run time software that takes advantage of the parallelism and manages it This allows the user to concern herself only with the expression of the algorithm and not with the expression of parallelism or the

implementation of it, which is required in explicitly parallel schemes. Explicit parallelism requires the user to explicitly manage the parallelism and synchronization, which can be a time consuming and error prone activity Functional programming allows the programmer to ignore these matters.

A *determinate* program is one where a given set of inputs always produces the same set of outputs, regardless of machine configuration, machine load, scheduling policy, and so on. A [21, p 305] language is determinate if it satisfies the Church-Rosser property, in which the value of an expression is independent of the order in which its subexpressions are evaluated (*i.e.* the order of evaluation to arrive at the result is unimportant) This property guarantees the determinacy of functional programs [24, 7]. The inverse, non-determinism [21, p. 331] is program behavior that can't be predicted from the source text alone, but depends on circumstances at runtime. [2]

The reasons for determinacy in functional languages is because of their single-assignment convention and lack of side effects [23, p 61] A *side effect* is anything that persists after the evaluation of an expression produces a result [34] Examples of how side effects occur in imperative languages are given in appendix C

Determinacy can be extremely important in parallel programming. It implies [21, p. 328] that the meaning of a program does not depend on the underlying machine implementing it This is invaluable in parallel systems [23, p. 61] It means, for example, that programs can be written and debugged in a functional language on a sequential machine, and then the *same* programs can be executed on a parallel machine for improved performance. This facilitates debugging tremendously: when left to the programmer, as in an imperative parallel programming language, determinacy is not guaranteed. The programmer is responsible for inserting the appropriate synchronization statements to produce determinate behavior Any omission, however minor, can produce non-determinate results [29, p 1] :

> *[Leaving determinate behavior up to the programmer]* makes debugging extremely difficult—for a given input, the program may produce different outputs for different

[2] Note that *determinate behavior* does not necessarily mean *deterministic execution* [7, p. 2] (although it can) implicit parallelism in the language, varying machine configurations and machine load can cause the particular choice of schedule for parallel activities in a program to be non-deterministic. However, the *result* computed should not vary with the schedule, for determinate behavior.

> machine configurations and/or scheduling policies, and this behavior may not immediately be obvious. Such timing-dependent errors may not even be reproducible in a debugger *[because statements inserted to debug may "disturb the experiment"]*

The last comment is important: using a debugger on a parallel program may cause timing-dependent errors responsible for non-determinate behavior to disappear. A functional program, because it is determinate, will avoid these pitfalls. Removing side effects is a crucial factor [23, p. 61]; the importance of minimizing side effects in a parallel system is intensified significantly, due to the careful synchronization required to ensure correct behavior when side effects are present.

The final characteristic important for parallel execution of functional languages is their [1, p. 15] equivalence of instruction scheduling constraints with data dependencies. This means that no artificial data dependencies are introduced. Artificial data dependencies in imperative languages are a consequence of being able to reassign new values to previously-defined variables. This is of course prohibited in functional languages. Thus, [1, p. 17] freedom from side effects is necessary to ensure that the data dependencies are the same as the sequencing constraints, and this is the case in functional languages. A review of data dependence types is presented in appendix D.

Thus, the key concepts introduced in this section are that functional programming languages are good for:

high-level programming: because of they support higher-order functions and a programming style close to the mathematical specifications for algorithms

determinate results: because of the Church-Rosser property, functional programs are guaranteed to give the same outputs, given the same inputs, irrespective of run-time conditions

parallel execution: because of the single assignment feature of functional languages that remove side effects with unknown consequences (which preclude parallel execution), and prohibit artifical data dependencies such as antidependencies and output dependencies.

C. Side Effects in Imperative Languages

A simple example of a function with a side effect is the following:

```
counter := 0;

function SquareAndCount(A: integer);
begin
   counter := counter + 1;
   return A^2;
end;

(1) b := SquareAndCount(2);
```

Function `SquareAndCount` side effects the variable `counter`, while returning the square of its argument. In imperative languages, the most common cause of side effects is the assignment statement, whose effect is to alter the underlying implicit store on which the imperative language is based so as to yield a different binding for a particular variable [24, p. 361]. However, functional languages prohibit the reassignment of a previously declared expression. no side effects (such as those caused by an assignment statement) are permitted. Without side effects, there is no way for concurrent portions of a program to affect one another adversely—this is simply another way of stating the Church-Rosser property [23, p. 61][1].

[1] There is a large body of compiler work devoted to the analysis of *interprocedural side effects* in imperative languages, which allow compilers to be more aggressive in scheduling different subroutines in parallel. See, for example, [13].

The most important area of concern for scientific parallel programmers in dealing with side effects is the way they influence the parallelization of operations on arrays, a data structure heavily used in scientific programming. Problems arise because of the way imperative languages treat data structures, as modifiable entities. The functional solution is to manipulate data structures in the same way scalars are treated, that is, as unmodifyable values, not as modifiable areas of memory, which is just a reflection of the von Neumann architecture on which imperative languages are built [1]. For example [1],

```
procedure SORT2(variable A: array[1..10] of real; J: integer);
var T: real;
begin if A[J] > A[J+1] begin
           T := A[J];
           A[J] := A[J+1];
           A[J+1] := T;
           end
end;

(1) SORT2(AA,J);
(2) SORT2(AA,K);
(3) P := AA[L];
```

since the values of J, K and L are not known at compile time, it must be assumed that statements (1), (2) and (3) will conflict if executed in parallel, and thus they must be executed sequentially, in the order specified. Another important problem is *aliasing*, where two different variable names refer to, in essence, the same memory location(s) [12]. This can occur when pointers to variables are used, or by using call by reference argument passing schemes in function calls, such as in the following [1]:

```
procedure REVERSE(var A, B: array[1..10] of real);
begin for J := 1 to 10 do
   B[J] := A[11-J];
end;
```

On first glance it would seem that all 10 assignments could be done concurrently, however, call by reference allows for the possibility of an invocation such as REVERSE(Z, Z), in which case concurrent execution would destroy the semantics of the above definition. Even if procedures and pointers are not used, the simple possibility of reassignment of an imperative array element makes parallelization uncertain.

```
(1) A[J]  := 3;
(2) X := A[K];
```

Both statements can proceed in parallel, unless J=K. Therefore, [1]

> If arrays exist as global objects in memory and are manipulated by statements and passed as pointers or procedure parameters, it is virtually impossible to tell, at the time an array element is modified, what effects that modification may have elsewhere in the program,

or, in other words, what side effects that array modification will produce.

D. Data Dependence Types

Let us examine what types of dependencies can occur. Refer to figure D 1. Data dependence relations are used to determine when two operations, statements, or two iterations of a loop can be executed in parallel [31] . In imperative languages (languages with side effects), three types of dependencies are found.

true or **flow dependence:** when two statements (such as S_1 and S_2, at left in figure D 1) cannot be executed at the same time since S_2 uses the value of A computed by S_1.

antidependence: (in the center, in figure D 1) since S_1 is to use the "old" value of B, it must be executed before S_2, and thus can't be executed in parallel.

output dependence: (at left, in figure D 1) if S_1 is executed after S_3, then A will contain the wrong value after this program segment They must therefore be executed in sequence.

As mentioned above [31, p. 1193], output dependencies and antidependencies are, in some sense, false dependencies. They arise not because data are being passed from one statement to another, but because the same memory location is written to in more than one place Functional languages prohibit this sort of behavior Imperative languages, however, [24, p 361] as a result of having implicit state that is modified (side effected) by commands generally have a notion of sequencing (of the commands) to permit precise and deterministic control over the state In functional languages, sequencing is theoretically constrained only by true dependencies, which is the minimum possible

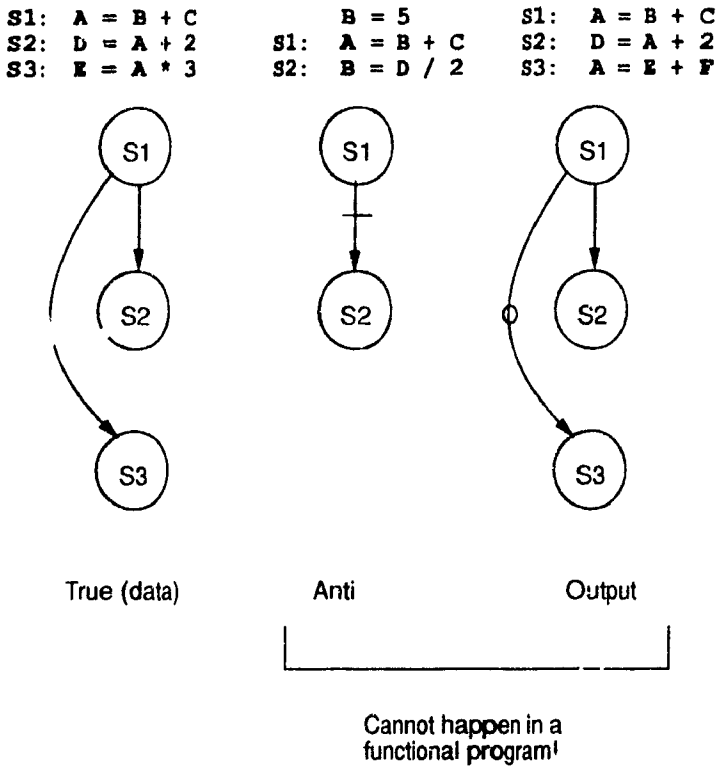


Figure D.1: Data dependence types in an imperative language.

constraint on parallel execution [1].

[1]There are compile-time analysis techniques that can often remove output and anti-dependencies, such as *variable renaming* and *node splitting*; see [31].

BIBLIOGRAPHY

[1] William B Ackerman Data flow languages. *IEEE Computer*, pages 15–23, February 1982.

[2] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, pages 26–34, August 1986.

[3] Stephen J Allan and R R. Oldehoeft. Hep sisal: Parallel functional programming. In *Parallel MIMD computation: The HEP supercomputer and its applications(?)*, pages 123–150. MIT Press(?), 1985(?).

[4] Arvind and David E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[5] Arvind, David E Culler, and Gino K. Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. In *Proceedings Supercomputing '88*, pages 60–69. IEEE Computer Society and ACM SIGARCH, IEEE Computer Society Press, 1988

[6] Arvind and Kattamuri Ekanadham. Future scientific programming on parallel machines. *Journal of Parallel and Distributed Computing*, 5:460–493, 1988.

[7] Arvind and Rishiyur S Nikhil. Executing a program on the mit tagged-token dataflow architecture. Computation Structures Group Memo 271, Laboratory for Computer Science of the Massachusetts Institute of Technology, March 1987.

[8] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. *Id Nouveau Reference Manual Part II: Operational Semantics*. Laboratory for Computer Science of the Massachusetts Institute of Technology, July 1988.

[9] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[10] BBN Advanced Computers Inc., 10 Fawcett St., Cambridge, MA 02238. *Inside the GP-1000*, 1988

[11] BBN Advanced Computers Inc., 10 Fawcett St., Cambridge MA 02238, U.S A. *Programming in C with the Uniform System*, 1.0 edition, October 1988.

[12] Micah Beck and Keshav Pingali. From control flow to dataflow. Technical Report TR 89-1050, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, October 1989

[13] David Callahan and Ken Kennedy Analysis of interprocedural side effects in a parallel programming environment *Journal of Parallel and Distributed Computing* 5.517–550, 1988

[14] Nicholas Carriero and David Gelernter. How to write parallel programs. A guide to the perplexed. *ACM Computing Surveys*, December 1989.

[15] David E. Culler and Gregory M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10 289–308, 1990.

[16] Boleslaw Szymanski *et al.* Conclusion. In Szymanski [36], chapter 9, pages 393–409

[17] Frank P. Ferrie and Peter Whaite, 1991. Personal communication.

[18] Martin A. Fischler and Oscar Firschein. Readings in computer vision. Issues, problems, principles, and paradigms. In Martin A. Fischler and Oscar Firschein, editors, *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*. Morgan Kaufmann Publishers, Inc., 1987.

[19] Daniel D. Gajski and Jih-Kwon Peir. Essential issues in multiprocessor systems. *IEEE Computer*, pages 9–27, June 1985.

[20] G.R. Gao, 1991. Personal communication.

[21] David Gelernter and Suresh Jagannathan. *Programming Linguistics*. The MIT Press, 1990

[22] A. Giordano, E.I Noviello, C. Sanges, and R. Vaccaro. A multigranularity massively parallel architecture for image understanding. In V. Cantoni, L.P. Cordella, S. Levialdi, and G. Sanniti di Baja, editors, *Progress in Image Analysis and Processing*, pages 742–750. World Scientific, 1989

[23] Paul Hudak. Para-functional programming. *IEEE Computer*, pages 60–69, August 1986

[24] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[25] Paul Hudak. Para-functional programming in haskell. In Szymanski [36], chapter 5, pages 159–196

[26] James E Narem Jr., 1991 Personal communication.

[27] Rishiyur S Nikhil. Id (version 88 0) reference manual. Computation Structures Group Memo 284, Laboratory for Computer Science of the Massachusetts Institute of Technology, March 1988.

[28] Rishiyur S Nikhil and Arving. *Programming in Id: A Parallel Programming Language.* Massachusetts Institute of Technology, August 1990. Draft of a book in preparation.

[29] Rishiyur S. Nikhil and P R. Fenstermacher. *Id World Reference Manual (for Lisp Machines).* Laboratory for Computer Science of the Massachusetts Institute of Technology, July 1988.

[30] Thomas J. Olson, Liudvikas Bukys, and Christopher M. Brown. Low-level image analysis on an mimd architecture In *Proceedings First International Conference on Computer Vision,* pages 468–475 The Computer Society of the IEEE, IEEE Computer Society Press, June 8-11 1987.

[31] David A Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM,* 29(12):1184–1201, December 1986.

[32] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer,* pages 13–23, December 1990.

[33] William H. Press, Brian P Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[34] Anthony Ralston and Edwin D. Reilly, Jr., editors. *Encyclopedia of Computer Science and Engineering* Van Nostrand Reinhold Co., second edition, 1983.

[35] Vivek Sarkar *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, 1989

[36] B. Szymanski, editor *Parallel functional languages and compilers.* ACM Press, 1991.

[37] S L. Tanimoto Architectural issues for intermediate-level vision. In M. J. B. Duff, editor, *Intermediate Level Image Processing,* chapter One, pages 3–17. Academic Press, London, 1986.

[38] Shreekant Thakar, Paul Gifford, and Gary Fielland. The balance multiprocessor system. *IEEE Micro,* 8:57–69, February 1988.

[39] P. Whaite and F.P. Ferrie. From uncertainty to visual exploration. In *Proceedings Third International Conference on Computer Vision*, pages 690–697, Los Alamitos, California, December 1990. IEEE Computer Society, IEEE Computer Society Press.

INDEX