Abstract. This paper outlines the beginning of a collaborative research effort whose overall goal is the development of an effective parallel programming environment called *ParLance*, consisting of a para-functional language, highly-optimizing compiler, parallel evaluation kernel, and interactive support for parallel program development. ParLance is intended to be relatively machine independent, with implementations planned for both a shared memory (ENCORE MultiMax) and a distributed-memory (INTEL iPSC) multiprocessor. Other machines to be considered include the Connection Machine and the WARP at CMU.

ParLance: A Para-Functional Programming Environment for Parallel and Distributed Computing

Paul Hudak[1], Jean-Marc Delosme[2], Ilse C.F. Ipsen[1]

Research Report YALEU/DCS/RR-524
March 1987

[1] Department of Computer Science, Yale University
[2] Department of Electrical Engineering, Yale University

The work presented in this paper was supported by the Army Research Office (DAAL03-86-K-0158), the Department of Energy (FG02-86ER25012), the National Science Foundation (DCR-8451415), and the Office of Naval Research (N000014-85-K-0461).

ParLance: A Para-Functional Programming Environment for Parallel and Distributed Computing

Paul Hudak Jean-Marc Delosme Ilse Ipsen

December 1986

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520

1 Introduction

A fundamental maxim guiding the design of high-level programming languages is the belief that languages should be pushed further in the direction of high-level abstractions, and less in the direction of machine detail. In other words, it is desirable for programmers to say less about the details of a particular computation, and more about the problem specification itself. As we move further in this direction, the dependence on smart compilation techniques becomes much greater, since the compiler has "further to go" in generating efficient code. At the extreme, the problem becomes completely intractable, since it reduces to synthesizing programs from totally abstract specifications such as those expressed using abstract data types or formal logic. Somewhere in between the extremes lies the world of practical programming languages and environments.

Functional (or *applicative*) programming languages constitute a class of high-level languages in which machine detail is minimized, yet whose programs can be executed with quite reasonable efficiency. However, the latter claim has only recently been substantiated. Early implementations were unbearably slow, and it has only been through diligent research in semantics-based inferencing techniques on one hand, and efficient evaluation strategies on the other, that we have arrived at the current state of affairs. It is now possible, for example, to write functional programs with arrays and achieve the same kind of efficiency as a similar but much lower-level Fortran program [36].

Functional programming languages are also well-suited to parallel computation, since their declarative nature does not place artificial constraints on the order of evaluation of expressions, and the "Church-Rosser Property" guarantees determinacy regardless of the execution order that happens to occur. These properties are simply manifestations of the high level of abstraction afforded by the languages. Stated another way, the concurrency in a functional program is "implicit" – there is no need for special constructs to express parallelism as is required in most conventional languages. This has been well-known since early work on dataflow machines, but in twenty years dataflow research has not made significant progress because the fine-grained parallelism in the dataflow model requires complex special-purpose hardware that is difficult and expensive to build [19].

Despite these difficulties, the functional programming paradigm is still a very good one for parallel computation. In fact, the first part of this paper is a summary of two research projects in Yale's Department of Computer Science that, quite independently and from two different perspectives, are based on the functional programming paradigm:

- *Para-functional programming* research is based on the use of functional languages for general-purpose computing, through smart compilation techniques, run-time scheduling, and user-supplied annotations to guide the operational aspects of parallel evaluation. This work has been carried out primarily by Paul Hudak in the Programming Languages and Systems Group (see Section 3).
- *Generation of optimal parallel implementations for systems of recurrence equations* is based on exploiting repetitiveness in certain classes of functional programs, and generating optimal processor assignments and schedules for those programs on particular multiprocessor configurations. This work has been carried out primarily by Ilse Ipsen and Jean-Marc Delosme in the Research Center for Scientific Computation (see Section 4.2).

These two independent research projects are in strong agreement philosophically, as both use functional programming as a basis for parallel computation.

The second part of this paper outlines the beginning of a collaborative research effort between these two groups, whose overall goal is the development of an effective parallel programming environment called *ParLance*, consisting of a language, highly-optimizing compiler, parallel evaluation kernel, and interactive support for parallel program development. ParLance is intended to be relatively machine independent, with implementations planned for both a shared-memory (Encore MultiMax) and distributed-memory (Intel iPSC) multiprocessor. Other machines to be considered include the Connection Machine and the WARP multiprocessor at CMU.

2 Motivation and Goals

The basic philosophy motivating our work can be summarized as follows: We believe that a programmer should be able to write parallel (or for that matter, sequential) programs in a very high-level language, as close to a true specification as possible, with no concern about operational issues such as evaluation order or task granularity. Given such an "executable specification," an optimizing compiler will be able to generate reasonably efficient code for a sequential machine, and in certain cases efficient code for a parallel machine, thus supporting the notion of "rapid prototyping" [28]. However, if this is not good enough, that is, if the program does not meet the performance requirements demanded of it, then the programmer should have ways to refine the operational behavior without restructuring the whole program or completely rewriting it in some other language – the same language syntax and semantics should be maintained throughout the entire software development process. Symbiotic with this goal is the desire for formal, rigorous models with which to reason about program behavior.

Our vehicle for accomplishing these goals is, of course, a functional language, for all of the reasons mentioned in the Introduction. Furthermore, finer control over operational issues such as evaluation order and process-to-processor mapping can be accomplished quite elegantly by *annotating the program with design information.* We refer to a functional language augmented with annotations as a *para-functional programming language.* Although the use of annotations may sound somewhat *ad hoc*, in fact annotations can be given a very precise formal semantics [33], consistent with the goals mentioned above. We prefer to think of the annotations as actually extending the semantic base of a language; they are annotations only in the sense that their removal results in a still perfectly valid functional program.

When viewed in the broader scope of software development methodologies, ParLance suggests the following software development scenario:

1. One first conceives of an algorithm and expresses it cleanly in a functional programming language. This high-level program is likely to be much closer to the problem specifications than conventional language realizations, thus aiding reasoning about the program, facilitating the debugging process, and in general allowing rapid prototyping of target systems.

2. Once the program has been written, it may be debugged and tested on either a sequential or parallel computer system. In the latter case, the compiler extracts as much parallelism as it can from the program, but with no intervention or awareness on the part of the user.

3. If the performance achieved in step two does not meet one's needs, portions of the program may be manually refined as follows:

- Some portions may be restructured so as to conform to the semantic constraints of a certain class of recurrence equations, in which case the compiler is able to generate efficient code with respect to specified performance constraints.
- Some portions may be annotated with design information that specifies certain operational behaviors desired by the programmer. These annotations can be made without affecting the program's functional behavior.

Both kinds of refinements can be combined freely within the same program.

There are three aspects of this methodology that we think significantly facilitate program development: First, the *functional* aspects of a program are effectively separated from most of the *operational* aspects. Second, the multiprocessor is viewed as a *single autonomous computer* onto which a program is mapped, rather than as a group of independent processors requiring complex communication and synchronization. And finally, sophisticated compiler technology is used to generate very efficient programs with respect to certain performance criteria for a reasonably broad class of problems.

Figure 1 shows the main components of ParLance and how they appear to a user of the system. From a research perspective, there are three aspects of ParLance that we wish to emphasize in this paper: (1) the para-functional language itself, including its formal semantics, (2) the automatic compilation strategies, based both on unconstrained lambda calculus (serial combinators) and affine recurrences (SAGA and CONDENSE), and (3) the ParLance programming environment, essentially combining the first two components into one integrated system. Work to date on the first two areas is described in the next two sections (3 and 4). Then in Section 5 we outline the research aimed at achieving the third goal, the development of the ParLance programming environment.

3 Para-Functional Programming Research

3.1 Annotations Viewed as Semantic Extensions

It is true that functional programs are fairly high-level (albeit executable) specifications, but as a result there are some operational issues, especially ones having to do with parallelism, that one cannot expect a compiler to be able to deal with optimally in all cases. One could give up on functional languages and resort to a lower-level parallel language such as Concurrent Pascal or CSP to express what one wants, but this would mean giving up all of the things that we like about functional languages. We argue that a better approach is to *annotate* functional programs with design information that expresses the operational behavior that is desired, and to design the annotations in such a way that they have no (or at least very little) impact on the functional behavior of a program.

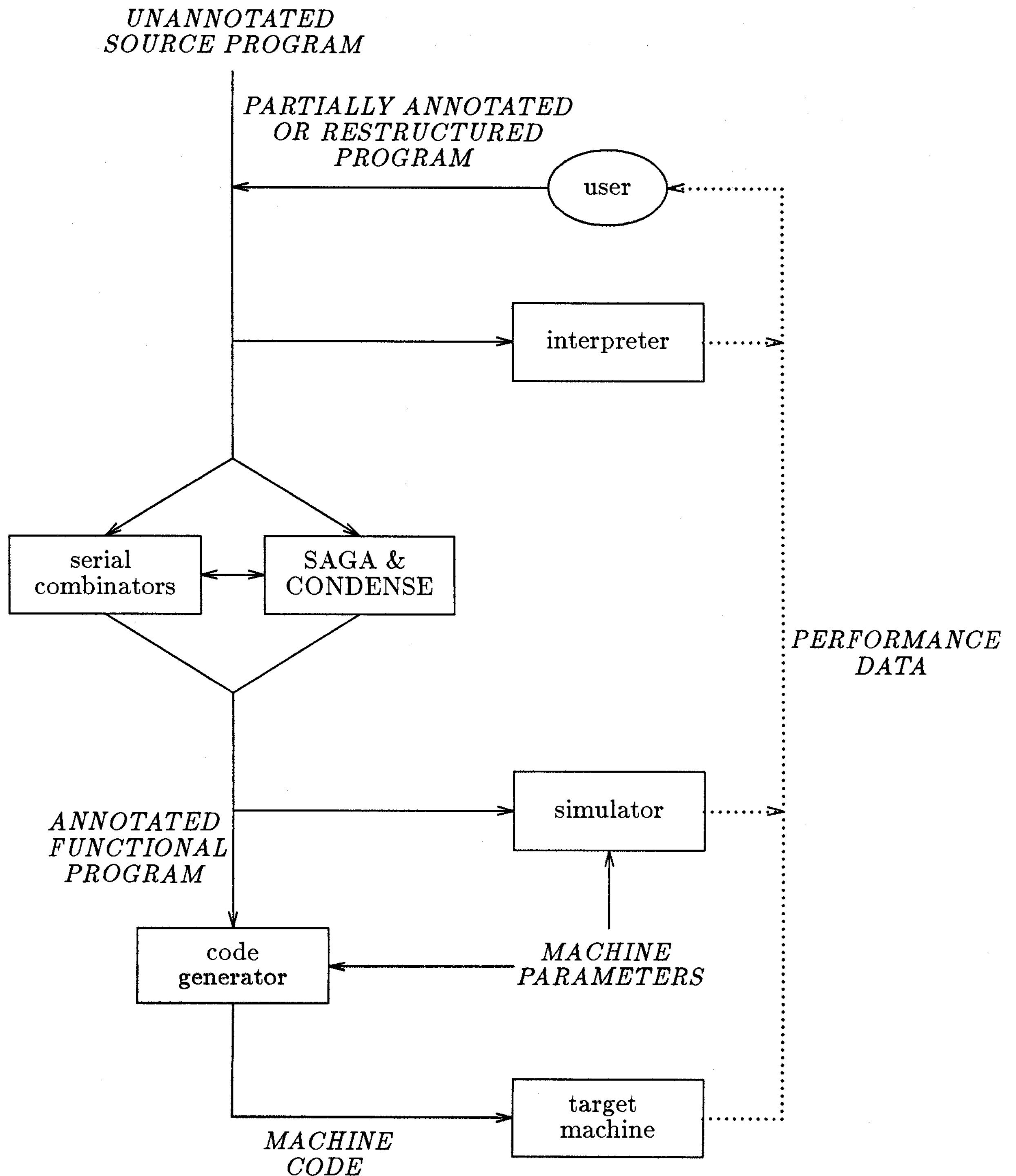

Figure 1: ParLance Compilation Strategy, Showing User Interaction

The separation of purely *operational* concerns that affect primarily efficiency, from purely *functional* concerns that affect the final answer, is what sets apart para-functional programming from most other approaches to parallel computation. By making this separation visible in the language itself (by expressing operational behavior using annotations), one is able to isolate, reason about, and thus debug, the two classes of problems more-or-less independently – this obviously facilitates the overall software development process.

On the other hand, annotations tend to conjure an image of ad hoc programming practice. But in fact, annotations can be given a very precise formal semantics [33], expressing a behavior that is "beyond" the functional behavior of the program.[1] We prefer to think of the annotations as actually *extending the semantic base* of a language; they are annotations only in the sense that their removal results in a still perfectly valid functional program.

Annotations to functional programs have previously been proposed in the context of both sequential and parallel computation, and include "memoizing" annotations [44], eager/lazy annotations [9,73], and prioritizing annotations [10]. At Yale an experimental para-functional language called ParAlfl is being designed that has several other kinds of annotations [31,35,42], the most important being those that express the *mapping* of a program onto a parallel machine – an operational concern that has nothing to do with the final value of the overall program. These annotations are described in more detail below.

3.2 Mapping Program to Machine

Part of the reason for providing mapping annotations is the realization that a compiler cannot be expected to determine the optimal mapping of program to machine in *all* cases (without actually executing the program first!), so it is desirable to be able to express the mapping *explicitly*. This need often arises, for example, in scientific computing, where many classical algorithms have been re-designed for optimal performance on particular multiprocessor topologies. Indeed, we have written a variety of programs for such algorithms in ParAlfl [35,42].

On the other hand, it *is* possible to automatically determine the optimal (or near optimal) mapping for a restricted class of functional programs – namely, the recurrence equations to be described in Section 4.2. This is a significant result, and one that fits in well with the para-functional programming paradigm – the use of smart compilation techniques to do the best job whenever possible, and the reliance on more explicit user control when automatic mappings become intractable. Moreover, the "output" of the transformations described in that section can be viewed as a para-functional program – i.e., a para-functional language simply becomes the "machine language" of that compilation

[1]The "para-" in "para-functional programming" is intended to convey this fact, as well as the fact that one of the primary motivations is *para*llel computation.

strategy. The relationship between para-functional programming and the ideas presented in Section 4.2 is discussed more later.

As an example of the use of mapping annotations, consider the expression $f(x) + g(y)$. The strict semantics of the "+" operator allows the two subexpressions to be evaluated in parallel. If we wish to express precisely *where* the subexpressions are to be evaluated, we may do so by annotating them, as in:

$$(f(x) \text{ on } 0) + (g(y) \text{ on } 1)$$

where 0 and 1 are assumed to be processor ids, or "pids."

Of course, this static mapping is not very interesting. It would be nice, for example, if we were able to refer to a processor *relative to the currently executing one.* We can provide this ability through the reserved identifier *self*, which when evaluated returns the pid of the currently executing processor. Using *self* we can now be more creative. For example, suppose we have a mesh or tree of processors that has a notion of "left" and "right"; we may then write:

$$(f(x) \text{ on } \mathit{left}(\mathit{self})) + (g(y) \text{ on } \mathit{right}(\mathit{self}))$$

to denote the computation of the two subexpressions in parallel on neighboring processors, with the sum being computed on *self.*

When used in the context of recursive programs, the use of *self* in mapped expressions can yield mappings to an arbitrary number of machines in arbitrary configurations. As a final example, here is a simple program to compute "parallel factorial," being representative of many divide-and-conquer algorithms:

$$\begin{aligned} pfac(lo, hi) = \; & \text{if } lo = hi \text{ then } lo \\ & \text{else if } lo = hi - 1 \text{ then } lo * hi \\ & \text{else let } mid = (lo + hi)/2 \\ & \quad\quad \text{in } (pfac(lo, mid) \text{ on } left(self))* \\ & \quad\quad\quad (pfac(mid + 1, hi) \text{ on } right(self)) \end{aligned}$$

$$\begin{aligned} left(pid) &= \text{if } 2 * pid > k \text{ then } pid \text{ else } 2 * pid \\ right(pid) &= \text{if } 2 * pid > k \text{ then } pid \text{ else } 2 * pid + 1 \end{aligned}$$

where $left$ and $right$ have been defined for a tree of k processors. If a different topology were used, $left$ and $right$ could be redefined accordingly, without changing the definition of $pfac$ at all; for example, it is relatively easy to embed a tree into a *hypercube.* In any case, if the pid of the root processor is *root*, then the factorial of n may be computed by the call $pfac(1, n)$ on *root.*

We have previously shown that our mapping annotations are sufficient to express a wide range of deterministic parallel algorithms in a concise and perspicuous manner [35,42].

Examples include several other divide-and-conquer algorithms; simple yet interesting programs such as several variations of a parallel fibonacci generator; numerical algorithms such as matrix multiplication, matrix-vector product, solving linear systems of block-matrices, and Jacobi's method; and systems programs such as distributed databases and resource management. Topologies considered include linear arrays, rings, trees, two-dimensional grids, tori, and hypercubes.[2]

Given two minor constraints, para-functional programs possess an important determinacy property, which can be stated as a theorem:

Theorem 1 (Determinacy) *A para-functional program in which (1) the identifier "self" appears only in pid expressions, and (2) all pid expressions terminate, is functionally equivalent to the same program with all of the annotations removed. That is, both programs return the same value.*[3]

The reason for the first constraint is that if it is violated, then once the mapping annotations are removed, all remaining occurrences of *self* will have the same value – viz., the pid of the "root" processor. Thus, removing the annotations might change the value of the program. For the simplest example of this, consider a system whose root processor has pid 0, and the expressions "*self on 1*" (whose value is 1) and "*self*" (whose value is 0). The purpose of the second constraint should be obvious: if the system diverges when determining the processor on which to execute the body of a mapped expression, then it will never get around to computing the value of that expression.

The work that is most similar in spirit to that presented here is Shapiro's "systolic programming" in Concurrent Prolog (CP) [72], whose mapping annotations were derived from earlier work on "turtle programs" in Logo [62]. However, there are several differences: (1) In CP (and LOGO) the "mapping" annotations are done by directing the process (or turtle) in a particular direction, and then advancing one processor (or turtle step) at a time. Our approach is more general – it allows arbitrary mappings to any processors in a purely functional way. (2) CP is based on a rather imperative reading of logic, LOGO on a conventional imperative language, and our work on a purely functional, declarative language. (3) There are no formal semantics for CP, LOGO or the annotations that they use; on the other hand, we have a precise denotational semantics for both our language and our mapping annotations.

[2]Mapping annotations are not particularly useful in a shared memory machine (other than to denote task granularity), unless the machine is heterogeneous, in which case it might be desirable to map a particular class of computations to a processor tailored for that class.

[3]A formal statement and proof of this property depends on a formal denotational semantics for the language, and may be found in [33].

3.3 Eager Expressions

A second form of annotation used in ParAlfl arises out of the occasional need for the programmer to "override" lazy evaluation, since normally an expression is not evaluated until absolutely necessary. This annotation would not be needed in a functional language with call-by-value semantics, such as pure Lisp, but lazy evaluation provides an extra degree of expressiveness that we desire. Thus an *eager expression* is introduced, which has the simple form "#*exp*" and intuitively forces the evaluation of *exp* in parallel with its immediately surrounding syntactic form. This annotation is similar to those used in [9,73].

A special case of eager computation occurs in the construction of *arrays*, which are almost always used in a context where the elements are computed in parallel. Because of this, the evaluations of the elements of an array are defined to occur eagerly (and in parallel, of course, if appropriately mapped).

Eager expressions are commonly used in lists. Consider, for example, the list expression $[x, \#y]$; normally lists are constructed lazily, so the values of x and y would not be evaluated until selected at some later time. But with the annotation shown, y would be evaluated as soon as the list is demanded. As with arrays, however, the expression does not wait for the value of y to return – instead it returns a partially constructed list just as it would with lazy evaluation.

The above discussion leads us to an important point about eager expressions: The *value* of an eager expression is the same as that of the expression without the annotation.[4] As with mapped expressions, the annotation only adds an *operational* semantics, and thus the user may invoke a non-terminating subcomputation yet have the overall program terminate. Indeed, in the above example y might not terminate, yet if only the first element of the list is selected for later use, the overall program may terminate properly. The "runaway process" that is computing y is often called an "irrelevant task," and there exist strategies for finding and deleting such tasks at run-time [5,24,34,41]. Indeed, given such an automatic "task collector" there are real situations where one might wish to invoke a non-terminating computation (an example of this is given in [42]).

3.4 Controlling Execution Order

Space consumption has traditionally been ignored in general-purpose functional programming environments, since large virtual memories, relatively large physical memories, and effective garbage collection strategies have lessened the severity of the problem. But if para-functional programming is to be expressive enough to deal with this problem, then

[4] Thus the determinacy theorem stated in the last section holds regardless of the presence of eager annotations.

language features need to be added to more precisely control the *evaluation order* of subexpressions. For example, if the invocation of a function f consumes space s, then in the evaluation of something like $f(x)+f(y)$, it may not be desirable to begin both invocations in parallel, since the total memory available might be less than $2s$. Indeed, it might be best to specify their evaluation order as being sequential and non-overlapping.

In the spirit of para-functional programming, this issue should be thought of as an *operational* concern, not a *functional* one; i.e., the same answer is returned in either case. Thus rather than restructuring the expression $f(x)+f(y)$ in some ad hoc way to achieve the desired result, what we seek is a language feature, like the mapping annotations, that can be layered over the program to achieve the desired effect. In ParAlfl we are experimenting with *synchronizing expressions* that allow the expression of explicit partial orders in subexpression evaluation. For the above example one could write:

$$\text{synch (ab) in a:}f(x)+\text{b:}f(y)$$

which says that the evaluation of the expression labelled a must precede that of the expression labelled b. The labels and the surrounding "synch" construct should be thought of as annotations, since the *value* of the whole expression is still $f(x)+f(y)$.

When used in a recursive setting, this technique can be used to do such interesting things as ensuring that two functions recurse "in lock-step." For example, consider the program:

$$\begin{array}{lcl} \text{synch (ab)} * \text{ in } f(l)+g(l) & & \\ \quad \text{where } f(lst) & = & \text{if } null(lst) \text{ then } nil \\ & & \text{else } \ldots \text{a:}f(tail(lst)) \ldots \\ \quad g(lst) & = & \text{if } null(lst) \text{ then } nil \\ & & \text{else } \ldots \text{b:}g(tail(lst)) \ldots \end{array}$$

Here the synchronizing expression "(ab)*" only permits zero or more evaluations of the expression labelled a followed by the expression labelled b.[5] Normally to achieve this kind of synchronization one would have to restructure the program by combining f and g into a single function that returned a composite result which was in turn decomposed for use where needed.

With respect to the work described in Section 4.2, synchronizing expressions are important in that they allow one to express the schedules generated by SAGA and CONDENSE. However, although synchronizing expressions are promising, more work is needed to determine if they are broadly expressive enough to solve *all* problems of this sort, and on the other hand to investigate the possibility of a simpler mechanism. We would also like to provide a formal semantics for such annotations – *partially-ordered multi-sets* are a promising vehicle for reaching this goal [63].

[5]Synchronizing expressions are similar to *path expressions* [25], and both are syntactically like *regular expressions* in automata theory.

3.5 Relationship to Recurrence Equations

As mentioned earlier, the framework within which the transformations to be discussed in the next section are developed is in strong agreement with the philosophy underlying para-functional programming. In particular, recurrence equations, taken literally, are functional programs. Their computabilty depends on the underlying domains and dependences, but in most cases they can be executed directly as functional programs after minor syntactic changes.

Furthermore, the purpose of the transformations to be described in the next section is to automatically infer the kinds of behaviors that annotations allow one to express explicitly. Thus we view a para-functional language as the "machine language" of those transformations. More specifically, inferred processor mappings and schedules can be expressed using mapped expressions and synchronizing expressions, respectively, as described in the previous paragraphs.

4 Automatic Compilation Techniques

Ideally we would like to be able to compile any annotation-free functional program into code for optimal execution on a given parallel machine. Unfortunately, it is fairly easy to show that determining the optimal processor assignment and evaluation schedule is undecideable in the general case. On the other hand, all is not lost – in particular, we have taken two specific directions in attacking this problem:

1. Given an arbitrary functional program, it is possible to perform a certain degree of program decomposition (using heuristics if necessary) that attempts to find the largest sequential threads of execution, called *serial combinators*, and execute them using a dynamic load-balancing strategy called *diffusion scheduling*. This work is described in Section 4.1.

2. Alternatively, there may be portions of the program whose data dependencies are sufficiently constrained so that optimal (or near optimal) decomposition and scheduling is possible. More specifically, this becomes possible when the semantic constraints of *affine recurrences* are met. This work is described in Section 4.2.

4.1 Serial Combinators and Diffusion Scheduling

Semantic Inferencing Techniques

Our work on serial combinators, using the full unconstrained lambda calculus as the source language to be compiled, depends critically on several inferencing techniques to

optimize programs for both parallel and sequential execution. Most of these efforts are examples of the use of *abstract interpretation* [13,59], a denotational semantics based inferencing methodology. Our work in this area has been fairly extensive, but without going into great detail it can be summarized as follows:

Strictness Analysis. Operationally speaking, *strictness analysis* is a way to determine that an argument to a function may be evaluated before (or in parallel with) a function call. Strictness analysis is essential for uncovering parallelism in languages based on lazy evaluation, which we prefer due to their expressive power. Previous strictness analyses [8,59] have been restricted to first-order languages or typed higher-order languages. Our contribution has been a way to infer strictness for languages with higher-order functions [43], and variations of strictness analysis to provide order-of-evaluation information about function arguments [7].

Copy Avoidance. Applicative data structures are mathematically elegant, but may imply a high space overhead if implemented in the obvious way. *Copy avoidance strategies* are concerned with eliminating this overhead, and we have developed both compile-time (using abstract interpretation) [36] and run-time [37] strategies. This work represents improvements over previous efforts [6,59,67,69], in that not only is the compile-time analysis more powerful, but in addition we have been targeting our work primarily toward the use of *arrays*, which are key data structures in our intended applications in scientific computing.

Applicative Data Flow Analysis. A *collecting interpretation of expressions* is an interpretation of a program that allows one to answer questions of the sort: "What are all possible values to which the expression *exp* might evaluate during program execution?" Answering such questions for functional programs is akin to traditional *data flow analysis* of imperative programs. We have developed such an interpretation for the full untyped lambda calculus with constants [32]. The method is simpler (no powerdomain construction is needed) yet more expressive than existing methods (such as Jones and Mycroft's *minimum function graph semantics* [46]). Indeed, our approach provides the first collecting interpretation for either lazy or higher-order programs.

Program Decomposition. An important problem in the automatic decomposition of a functional program for parallel evaluation is determining the largest possible "grains" of parallelism, which in turn requires "uncurrying" function applications. Doing so requires determining the *degree of sharing of partially applied functions*. We have solved this problem, again using the tools provided by abstract interpretation [20,22].

These abstract interpretation techniques set up the framework within which serial combinators are generated, as described in the remainder of this section.

Generating Serial Combinators

Of the many possible parallel computing models, it is probably safe to say that a fixed set of combinators offers in some sense the *finest* granularity of computation, even finer than dataflow. However, if one studies the performance figures for existing multiprocessors, it becomes apparent that the ratio of interprocessor communication time to CPU instruction speed is generally quite high; typically anywhere from 10 to 100. Thus it seems that relatively large "grains" of parallelism need to be found for our overall strategy to be successful. We initially became motivated to do this after observing via simulation that with a fixed set of combinators it is possible for a purely sequential computation (i.e., one whose data-dependencies preclude any parallelism) to become decomposed for execution on several processors [39]. Clearly this is a wasted effort, since such a decomposition can only add communication costs to an already-sequential computation!

Although considerable progress has been made in parallel evaluation models, little work has been done on choosing the right granularity for parallel program decomposition. Our goal is to retain the environment-less nature of combinators and their usefulness in a parallel graph reducer, while maximizing their granularity and ensuring that parallelism is not lost. The strategy used by Keller and Lin [49] relies predominantly on the functions defined by the user, and is similar to the "lambda-lifted" functions used by Johnsson [45] – neither technique guarantees "fully lazy" combinators, and no analysis is attempted within function bodies to detect parallelism. Super-combinators are a step in the right direction, having considerably larger granularity than a fixed set of combinators, but they can be made even larger, and are again targeted for sequential machines, thus not exploiting all opportunities for parallelism.

Serial combinators [40] are in some sense the best that one can do in the general case, since they have the following properties:

1. They are combinators, facilitating their use in a graph reduction machine (especially parallel ones).
2. They result in a fully lazy evaluation, guaranteeing that no extraneous computations are performed.
3. They have no concurrent substructure, guaranteeing that no available parallelism will be lost.
4. There are no larger objects having these same properties, ensuring that no extraneous communication costs are incurred because of too fine a granularity.

Even if the third refinement is ignored, simulations demonstrate that the resulting combinators are more efficient than super-combinators, on both sequential and parallel machines.

This is because serial combinators have a larger granularity, including the fact that they may be recursive, eliminating inefficiencies involving the **Y** combinator.

Despite the relatively simple characterization of serial combinators as given above, we make important pragmatic refinements that complicate their analysis. In particular, we take into consideration strictness properties of functions, common subexpressions, complexity of subexpressions, and the overhead for distributing a computation.

The overall translation scheme from source program to serial combinators consists of three phases. The source program is first translated into an "intermediate form" called a *normalized equation group.* A set of *refined super-combinators*[6] is then generated, having the improved properties mentioned earlier. Finally, issues of parallel computation are taken into account as the super-combinators are further refined into serial combinators.

Diffusion Scheduling

The scheduling of serial combinators for execution on a given multiprocessor is controlled by a dynamic load-balancing mechanism that we refer to as *diffusion scheduling*, which takes into account such factors as processor load, memory utilization, and direction of global references. The intent is for tasks to be "pushed away" from busy processors, and "drawn toward" those to which they have global references (thus maintaining locality). In this way work "diffuses" through the network in the direction of least resistance.

For example, the diffusion scheduler may choose a neighboring processor p on which to evaluate a serial combinator s, such that $C(s,p)$ is minimized, where:

$$C(s,p) = load(p) + (k * ref\text{-}cost(s,p))$$

and *ref-cost(s,p)* is a measure of the cost of p's references to nodes residing on processors other than p. *load*(p) is simply the number of tasks on p's task queue. The constant k is a weighting factor that indicates the relative importance of the *ref-cost* of n compared to the work load of p. The lower the value of k, the more likely it is that work simply diffuses towards the least-loaded processors.

An important optimization to this overall strategy is to avoid the message-passing protocol for local communications, bypassing the task queue entirely whenever possible. This, coupled with the fact that the combinator definitions are represented as *conventional compiled straight-line code*, means that we can take advantage of the efficient sequential features of the von Neumann processors, using message-passing only when needed. These optimizations are described in more detail in [20].

A diffusion scheduler for serial combinator evaluation, part of a virtual parallel graph reducer called *Alfalfa*, is currently being implemented on three commercial multiprocessors: an Intel iPSC hypercube, an NCube hypercube, and an Encore shared-memory machine

[6] Or, more whimsically, *super-duper-combinators*!

[20]. As of this writing the implementation on the Intel iPSC has just begun to work, and preliminary benchmarks look promising [21].

4.2 Efficient Compilation of Recurrence Equations

Numerous programs for scientific computation are devoted to computing *recurrence equations* [50]. Thus, efforts dedicated to the automatic generation of highly efficient parallel implementations of recurrence equations are well invested. On account of the repetitiveness of their computations we classify recurrence equations with respect to the relationships, or *dependences,* between their indices.

Frequently one encounters sets of recurrence equations in which these dependences are *not* functions of the actual *values* of the variables in the equations. Annotations representing schedules (specifications of time and processor for each operation) and interprocessor communications for such a set of equations may then be expressed as functions of the indices. The functions can be determined at compile time and evaluated at run time once the recurrence bounds and stride parameters are known. This *global* approach to scheduling results in parametrized annotations for the *entire* set of equations, and it is viable because of the weak relation between dependences and actual data values, i.e. the availability of almost complete scheduling information at compile time. On the other hand, when the dependences within a set of equations are functions of the data, it seems most effective to determine *local,* sequential components and distribute their execution across the processors at run time – this is the route followed in Alfalfa.

The global approach for generating annotations described in this section and the local approach of Alfalfa are thus two extremes in a spectrum defined by the amount of dependence information available at compile time. Once these two approaches have been fully developed we hope that we will have gained enough insight to enable us to develop approaches more suited to the intermediate cases, where local schedulers would dynamically schedule the execution subject to local data and global control information. We feel that the general problem of efficient "adaptive" scheduling (adaptive with respect to graded levels of uncertainty) is a promising area of future research for which our current work provides a strong basis.

In this section we shall consider recurrence equations whose variables have index functions that are independent of the actual values of the variables. Some recurrence equations, such as Fast Fourier Transform algorithms, have indices whose dependence is non-linear. A general theory for the implementation of recurrences with non-linear dependences does not appear feasible, and such recursions must be dealt with on a case by case basis. However, most recursions found in scientific computation have linear or, more accurately, affine dependences [23]. We believe that it is possible to develop procedures that generate efficient, parallel implementations of recurrence equations with affine dependences. We have already

identified the steps that such a procedure should follow, and we are currently designing fast algorithms for each of those steps.

To illustrate the notion of affine dependence consider finding the maximal element among n numbers $\{a_1, \ldots a_n\}$. This can be accomplished by constructing a heap:

$$h_i = a_{i-n+1}, \quad n \leq i \leq 2n-1, \qquad h_i = \max\{h_{2i}, h_{2i+1}\}, \quad 1 \leq i \leq n-1 \qquad (hp)$$

where h_1 contains the maximal element, or by performing a linear search:

$$s_1 = a_1, \qquad s_i = \max\{s_{i-1}, a_i\}, \quad 2 \leq i \leq n, \qquad (ls)$$

in which case s_n contains the maximal element. Both equations (hp) and (ls) are recurrence equations with *affine dependences:* every subscript is an affine function of the index i, that is, a linear function plus an additive constant. Equation (ls) belongs to the subclass of *uniform recurrence equations:* all the index functions are just translations, with linear part equal to the identity. Note that this classification depends only on the *indices* of the variables. No restriction is imposed on the functions, like "max " above, that compute the values of the variables.

It has been recognized for several years that *simple* yet highly efficient schedules can be found for the implementation of uniform recurrence equations on "systolic array" architectures. *Systolic arrays* are homogeneous networks of processors with local memory and extremely fast interprocessor communication [52,54]. The elegance of systolic array implementations led to a flourishing of literature in the early eighties describing implementations for basic linear algebra, signal processing and graph algorithms, see e.g. [1,26]. Although today most systolic implementations are still found by hand, progress has been made with regard to automating the parallel implementation of uniform recurrences [57,64,65]. It is the high degree of repetition and simple dependences between computations that makes feasible the automatic implementation of uniform recurrence equations: the search for efficient schedules may be restricted to *simple* schedules that exploit this structure (by *schedule* we mean the specification of time instances and processor assignments for the execution of individual operations). In most cases schedules defined by affine functions can be quickly found whose computation times are within a few per cent of the best attainable.

Most recurrence equations, though, turn out to be more complex than uniform recurrences. Often they consist of several sets of recurrence equations, called *steps*, in such a way that results of computations of one step represent input data for other steps. Moreover, the variables within a step do not necessarily have the same number of indices. Such equations belong to the larger class of recurrence equations with *affine dependences*; they cannot be handled by automatic implementation techniques for uniform recurrence equations. Thus, the limitations of existing techniques become apparent when automatization is needed most, that is, when hand design of efficient systolic implementations is nearly impossible. In order to overcome these limitations we have started to develop a method

that determines systolic implementations of algorithms with *affine* dependences, these implementations attain minimal computation time on a minimal number of processors [14,15,16,18]. The method is called *SAGA,* an acronym for *Systolic Array Generating Algorithms.* SAGA is a sophisticated generalization of the most advanced techniques devised for the implementation of uniform recurrences. Only with the help of SAGA, for example, was it possible to develop the algorithm and systolic implementation of the Toeplitz system solver in [17] whose efficiency is superior to all other existing Toeplitz solver implementations.

The number of processors in the systolic implementations determined by SAGA depends on the problem at hand. This is appropriate for many signal processing applications, where a systolic array always works on problems of the same size (for instance, the problem size might be determined by the number of antennae in an antenna array). In the context of scientific computation, however, different problems of different sizes all have to be implemented on the same parallel machine, and the results from SAGA are not directly applicable in this case, when the number of processors is fixed and determined in advance. This is why we are using a two-phase approach for the implementation of systems of recurrence equations on a given target architecture (the Parafrase restructuring compiler [74] employs a similar decomposition). In the first phase SAGA determines an optimal systolic array implementation that is essentially target-architecture independent. In the second phase, called *CONDENSE,* the processors of this array are partitioned into groups so that each group is assigned, or *condensed,* to one processor of the target architecture. The condensation is done so as to minimize the total run time, and is guided by the systolic array from SAGA.

The two-phase approach, SAGA followed by an architecture-specific phase, is applicable to systolic architectures such as the ten-processor CMU WARP [2,3], where partitioning techniques for systolic arrays [58] would provide a starting point for the second phase. For some other architectures, such as the Connection Machine [29], the second phase is almost trivial and SAGA can be applied directly to find optimal implementations. The class of architectures of particular interest to us consists of homogeneous networks of processors with local memory where communication takes place via message-passing and the processors do not have fast access to a large common memory. Examples are hypercube and mesh architectures such as the Intel iPSC, the Caltech Cosmic Cube [70] or a mesh of MOSAIC processors connected via Torus Routing Chips [56].

We will end this section with a brief summary of the features of SAGA & CONDENSE. First of all, in abidance with functional language principles, SAGA & CONDENSE make it possible to express algorithms in a natural fashion. For instance, to compute an inner product

$$\sum_{i=1}^{n} a_i b_i \qquad (ip)$$

or part of a dynamic programming algorithm

$$\min_{i \leq k \leq j} \{ f(a_{ik}, a_{kj}) \} \qquad (dp)$$

the user does not specify the *order* of evaluation of these functions; SAGA will automatically find an order that does not introduce constraining dependences.

The implementations found by SAGA and adapted to a specific architecture by CONDENSE are *systolic* implementations. This does not constitute a severe restriction. As illustrated in equation (hp) systolic implementations include trees and support "logarithmic" evaluation schemes, and broadcasting in general; their data communication structure is as repetitive and simple as their computational structure; and, due to the systolic nature, data "flows" through the array so that the storage per processor is kept low. In fact, the systolic algorithm and programming approach has already shown the best performance on hypercube architectures with regard to important numerical applications [66] and has been advocated [71] as an "algorithm design and programming methodology for general purpose [...] parallel computers" due to its wide applicability, convenient scalability and ease of programming.

SAGA generates minimal-time systolic implementations. Not insisting on minimal computation time can easily result in implementations that are fifty percent or possibly an order of magnitude slower than the best implementation. Once minimal computation time is attained, SAGA minimizes the number of processors, often improving efficiency by a factor of two or three. Typically SAGA generates several optimal arrays, of one, two or higher dimensions (for instance, it generates one-, two- and three-dimensional arrays for a two-dimensional convolution). The constraints in SAGA's optimization problems are not very sensitive to the target architecture; this low sensitivity is an attractive aspect of our two-phase decomposition. SAGA is invoked at compile time since it does not require knowledge of the exact bounds in the recurrence equations.

CONDENSE employs a collection of heuristic strategies that quickly determine, at run time, processor partitionings optimized with regard to the parameters of the target architecture. In particular, for each of the arrays generated by SAGA the size and aspect ratios of processor blocks and the packet sizes for data transfer are determined, and the one with the best run time on the target architecture is selected. Ample flexibility with regard to the properties of the target architecture, such as processor interconnection topology, number of processors, and communication model and costs, is achieved by adjusting the parameters within CONDENSE. Accordingly, machines like the Intel iPSC and the Connection Machine that have such different communication properties can be accommodated in our framework.

Description of SAGA SAGA requires as input a *single assignment language* [4], a condition that is fulfilled naturally in a functional language. As a result the only dependences

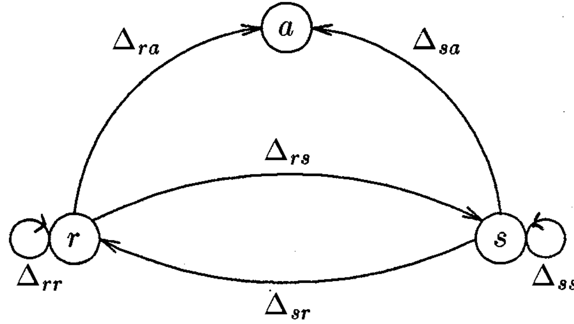

Figure 2: Dependence Graph for Example Equations (*ex*)

are the *data dependences* induced by the recurrence equations; there are no extraneous *output* and *anti dependences* such as found in imperative languages [51,60]. The goal of SAGA is to generate simple *control dependences* in such a way that the overall run time is kept minimal. The following description attempts, where applicable, to relate SAGA's features to the better known notions used in compiler design.

By exploiting the computational regularity inherent in systems of recurrence equations it is possible to design fast algorithms for SAGA that determine parallel implementations whose run time is *provably* minimal (asymptotically).

The regularity in a system of *recurrence equations,* such as

$$\begin{array}{ll} r_{i0} = a_i, & 1 \le i \le n \\ s_{i1} = a_{i-2}, & 3 \le i \le n+2 \\ r_{i,j} = r_{i-1,j-1} - s_{i+1,j}, & 1 \le j \le n-1, \quad j+1 \le i \le n, \\ s_{i,j} = -r_{i-2,j-2} + s_{i,j-1}, & 2 \le j \le n, \quad j+2 \le i \le n+2, \end{array} \qquad (ex)$$

is captured in a *dependence graph*, shown in Figure 2. The nodes of the dependence graph correspond to variable names (in an imperative language they might also correspond to program statements [51,60]). The arcs of the graph correspond to the *data* [60] or *flow* [51] *dependences*: there is an arc (r, s) from variable r to variable s since the computation of r requires s. Each arc is labeled by the corresponding *dependence mapping*

$$(i \quad j)\,\Delta_{rr} = (i-1 \quad j-1), \qquad (i \quad j)\,\Delta_{rs} = (i+1 \quad j),$$
$$(i \quad j)\,\Delta_{sr} = (i-2 \quad j-2), \qquad (i \quad j)\,\Delta_{ss} = (i \quad j-1),$$

whose argument is the index $(i \quad j)$ of the variable on the left-hand side of the equation and whose value is the index of the used variable on the right-hand side.

The *domain of computation* of r is the domain of the dependence mappings Δ_{rr} and Δ_{rs}

$$C_r = \{(i,j) : j+1 \leq i \leq n,\ 1 \leq j \leq n-1\},$$

and, similarly, the domain of computation for s is

$$C_s = \{(i,j) : j+2 \leq i \leq n+2,\ 2 \leq j \leq n\}.$$

In this example if the domains for Δ_{rr} and Δ_{rs} had not been identical, the renaming procedure in [18] would have been applied first. The notion of "domain" thus replaces that of "loop dependence" [51].

The equations in the example represent *uniform recurrences* since all the dependence mappings are translations such as

$$P\Delta_{rs} = PD_{rs} + d_{rs},$$

where

$$P = (i \quad j), \qquad D_{rs} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I, \qquad d_{rs} = (1 \quad 0).$$

The time step t at which the quantities r_P and s_P are computed for a point P belonging to the domains C_r and C_s can be simply expressed as [57,64]

$$t = P\tau + u. \tag{1}$$

Here, $\tau = (\tau_1\ \tau_2)^T$ is a column vector, called the *time vector* and u is a scalar offset. The time of computation is therefore defined by an *affine* time function.

The dependences restrict the choice of the time vector in the sense that the computation of r_P, or s_P, must not use a value that has not yet been computed. Hence τ should satisfy

$$(1 \quad 1)\tau > 0, \quad (-1 \quad 0)\tau > 0, \quad (2 \quad 2)\tau > 0, \quad (0 \quad 1)\tau > 0. \tag{2}$$

Among the many vectors satisfying (2) one selects those τ that minimize the computation time. The shapes of the domains C_r and C_s together with the dependences provide the first and the last computed index, and thus an expression for the computation time in the following optimization problem:

Determine τ that satisfies (2) (i.e., $0 < -\tau_1 < \tau_2$) and minimizes

$$(n+2 \quad n)\tau - (n+2 \quad 1)\tau = (0 \quad n-1)\tau.$$

This problem has the unique solution $\tau = (-1 \quad 2)^T$ and results in a computation time of $2n$ steps. However, the longest chain of dependent computations traverses only

n points, thus the above time is far from optimal. The computation time can actually be made optimal in such cases without increasing the complexity of the implementation, by using *different* offsets, u_r and u_s, for *different* variables r and s [16,65]. Geometric insight is gained by viewing such a time schedule as the application of a *single* affine function, with the same offset, to both variables r and s but the domains of r and s are *displaced* with respect to each other [16]. The idea here is that the original system of recurrence equations (ex) is written in a somewhat arbitrary fashion and a better formulation may be obtained by translating the domain of s with respect to the domain of r. A translation by $(-1 \quad -1)$ results in the equivalent system of equations

$$\begin{array}{lll} r_{i0} = a_i, & 1 \le i \le n & \\ s_{i0} = a_{i-1}, & 2 \le i \le n+1 & \\ r_{i,j} = r_{i-1,j-1} - s_{i,j-1}, & 1 \le j \le n-1, & j+1 \le i \le n \\ s_{i,j} = -r_{i-1,j-1} + s_{i,j-1}, & 1 \le j \le n-1, & j+2 \le i \le n+1 \end{array} \qquad (ex')$$

which, for the affine time function (1) with $\tau = (0 \quad 1)^T$, gives the minimal computation time, of n steps.

The question remains how the displacement $(-1 \quad -1)$ of C_s with respect to C_r that leads to the optimal computation time was discovered. The answer, according to our approach, is obtained by considering the *invariants* associated with the system of recurrence equations. In the above example, the invariants are those translations that are associated with the composition of dependence mappings around "cycles": Δ_{rr}, Δ_{ss}, $\Delta_{rs}\Delta_{sr}$ and $\Delta_{sr}\Delta_{rs}$; the time vector τ should be consistent with the invariants

$$(1 \quad 1)\tau > 0, \quad (0 \quad 1)\tau > 0, \quad (1 \quad 2)\tau > 0. \qquad (3)$$

Naturally, the invariants are the same for the original system (ex) and the "displaced" system (ex'). The cone defined by the conditions (3) ($\tau_1+\tau_2 > 0$, $\tau_2 > 0$) is called the "time cone" for the recurrence equations. Any vector τ within the time cone defines a general "global" time function for the system of recurrence equations and one can always find translations that define precise "local" time functions for each of the computed variables; this means, one can always find translations of the domain of s with respect to the domain of r so that $t = P\tau + u$ is a time function consistent with the (new) dependences d_{rr}, d_{rs}, d_{sr} and d_{ss}. Intuitively, a global time function provides only enough information about the execution time of each operation to guarantee an *asymptotic* time bound, while a local time function specifies the *exact* time of each operation.

Thus, from the original recurrence equations (ex) our procedure first computes the invariants that define the time cone. From the time cone a time vector τ is then selected that minimizes the "global" computation time $\max\{(0 \quad n)\tau, (n \quad n)\tau\}$, i.e., τ minimizes $\max\{(0 \quad 1)\tau, (1 \quad 1)\tau\}$ subject to $\tau_1 + \tau_2 > 0$, $\tau_2 > 0$, resulting in $\tau = (0 \quad 1)^T$. Finally, the translation is chosen to satisfy secondary criteria such as minimization of register count

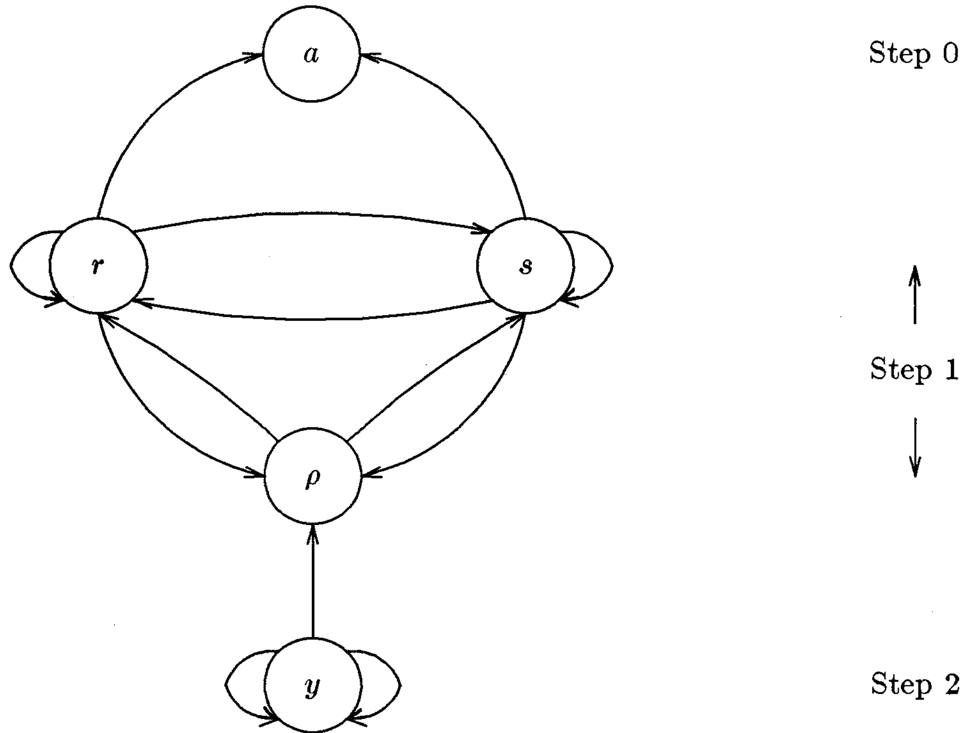

Figure 3: Dependence Graph for the Toeplitz Factorization Algorithm

and communication. This results in $(-1 \quad -1)$ as the translation of the domain of s, thus yielding the modified system (ex'). The determination of τ represents the "global" optimization problem while the determination of the displacement represents what we call the "local" optimization.

These concepts may be applied to more general recurrence equations such as the following algorithm, which computes for a given symmetric Toeplitz matrix both its LDL^T

factorization and the LDL^T factorization of its inverse

$$\begin{array}{ll} r_{i,0} = a_{i-1}, & 1 \le i \le n \\ s_{i,0} = a_{i-1}, & 2 \le i \le n \\ \rho_j = s_{j+1,j-1}/r_{j,j-1}, & 1 \le j \le n-1 \\ r_{i,j} = r_{i-1,j-1} - \rho_j s_{i,j-1}, & j+1 \le i \le n \\ s_{i,j} = -\rho_j r_{i-1,j-1} + s_{i,j-1}, & j+2 \le i \le n \end{array} \qquad (f)$$

$$\begin{array}{ll} y_{1,0} = 1 & \\ y_{i,i-2} = 0, & 2 \le i \le n \\ y_{i,j} = -\rho_j y_{j+2-i,j-1} + y_{i,j-1}, & 1 \le j \le n-1, \quad 1 \le i \le j+1 \end{array}$$

In such a general case, SAGA proceeds as follows:

1. Decomposition of the algorithm into steps.

   SAGA decomposes the set of recurrence equations into tightly coupled systems of equations referred to as "steps" by determining the *strongly connected components* of the dependence graph (π-blocks in [60]), as shown in Figure 3. Consequently, results from one step that are required as inputs to a second step do not depend on results from the second step. The decomposition of the algorithm into steps provides a unique division of the implementation problem into (smaller) subproblems for the steps that can be solved independently and whose solutions can then be combined into an optimal solution for the whole without disregarding potential designs.

2. Separate computability analysis and characterization of *all* possible schedules for each step.

   A proper parallel implementation of recurrence equations necessitates a preliminary analysis of the computability of the equations (likewise, the first phase in the design of a control system is a controllability analysis). The information gathered during the verification phase is then used in the time scheduling and processor assignment phase. The notion of time cone is generalized to apply to steps with affine dependences such as $\Delta_{r\rho}$ in equations (f) above.

   It often occurs that the inherent parallelism in an algorithm is limited (for instance, if the dependences in (ex) were changed in such a way that the longest chain of dependent computations traverses $O(n^2)$ points instead of n points.) Although the time cone is empty in this case one can still find simple schedules that minimize the computation time. In the case of uniform recurrence equations the methods in [57,64] break down. However, the seminal work of Karp, Miller and Winograd in 1967 [47,48] provides the essentials of a solution to the associated time scheduling problem. Our theory parallels that work at a fundamental level for affine dependences.

3. Solution of the global optimization problem to find the global schedule that minimizes the total computation time and the number of processors.

4. Separate solution of the local optimization problems to find the local schedules for each step.

SAGA synthesizes systolic arrays through the direct determination of transformations that are applied to each step as a whole. This approach is computationally efficient and requires interaction only at a high level. Approaches based on incremental transformations [30,53] appear less efficient and call for low level interactive decisions; some decisions may be avoided through the use of greedy algorithms [12] but there is no proof of the optimality of the resulting arrays. SAGA performs these decisions automatically since the knowledge required to make them is encoded into a hierarchy of objectives constructed before the invocation of SAGA. SAGA's efficiency in finding optimal implementations is a key feature for integrating it into a general parallel compiler.

Description of CONDENSE Some of the issues dealt with by CONDENSE have already been encountered in the context of efficient register management and memory access on vector register machines (see the *loop distribution* and *loop blocking* strategies in [51]). In order to determine the total run time of a systolic implementation from SAGA on the target architecture, fast algorithms compute optimal condensation parameters such as the dimension of the condensed array and its data transfer parameters, e.g. the packet size. Our model is very general and applicable, for instance, to systolic implementations encompassing streams of data flowing at different speeds, as in systolic implementations of Gauss-Jordan and dynamic programming algorithms. When incorporated into an actual code generator, the expressions for the total run time of the systolic implementation on the target architecture are set up at compile time and their optimal solution is determined at run time.

Differences Between Our Approach and Conventional Code Improvement Techniques In the remainder of this section we illustrate the superiority of our approach over the well known code optimization, or more accurately code improvement, techniques.

- *Repetitive computations with limited data dependence.* Only by restricting the range of SAGA to this particular class of computations, can we claim optimality and achieve mathematical tractability. However, it is exactly this class of computations which prevails in scientific computing and ensures the wide applicability of SAGA. We assume that recurrence equations consist of data independent statements; data dependent statements in scientific calculations (such as "if" or "while") can be handled without difficulty: they occur either for the prevention of under- and overflow (such as if-statements in the implementation of plane rotations, see [23]) and can be relegated to the processor design level [1,26] or they occur when testing convergence

(such as while statements in the unshifted QR-algorithm for the symmetric tridiagonal eigenvalue problem) in which case one iteration is implemented in parallel and a whole sequence of iterations is "pipelined in time" [27,68].

- *Thorough parallelism detection.* As opposed to the *code improvement* transformations of a compiler SAGA and CONDENSE really perform *optimization* transformations. Compilers for sequential as well as parallel languages [61] choose among a few ways of evaluating nested loops (usually interchanging or unrolling the loops). The concept of "time cone", a notion borrowed from multi-dimensional systems theory [55], allows SAGA to summarize *all* possible schedules and to choose among those the ones with minimal complexity that yield minimal computation time.

- *Global (and local) optimization.* All compiler optimizations occur at a local level, that is the compiler tries to efficiently schedule a nest of loops but does not look far beyond. We have proved that the problem of finding minimal-time implementations for systems of recurrence equations can be decomposed into a sequence of subproblems without losing optimality: SAGA first finds all the schedules for each step, then solves a global optimization problem to select step schedules whose composition yields minimal computation time with a minimal number of processors.

- *Accounting for processor topology.* Hitherto, schedulers for parallel architectures have been restricted to shared memory machines (see Parafrase [74]), and have disregarded processor interconnection topology. The only work on scheduling that takes into account topology is scheduling on linearly connected processor arrays [11] which is not practical for recurrence equations since it employs the computation graph instead of the dependence graph, and poses restrictions on the computation graph that could lead to artificial data dependences.

5 The Incarnation of Parlance

In this section we outline a recently begun project whose goal is to combine the research described in the previous two sections into a single integratred parallel programming environment called *Parlance.* As mentioned in the introduction, ParLance will consist of a language, highly-optimizing compiler, parallel evaluation kernel, and interactive support environment, with implementations planned for both a shared-memory (Encore MultiMax) and distributed-memory (Intel iPSC) multiprocessor.

5.1 Parallel Implementation Issues

In recent years great advances have been made in implementing functional languages for both sequential and parallel machines, and much of that work is applicable here. In

particular, *graph reduction* provides a very natural way to coordinate the parallel evaluation of subexpressions, and solves problems such as how to migrate the values of lexically bound variables from one processor to another [38,40]. As mentioned earlier, a virtual parallel graph reducer called *Alfalfa* is currently being implemented at Yale on three commercial multiprocessors: an Intel iPSC hypercube, an NCube hypercube, and an Encore shared-memory machine [20,21]. This graph reduction engine will be able to support both implicit (dynamic) and explicit (annotated) task allocation.

We have also designed a para-functional programming language called *ParAlfl* that incorporates many of the ideas mentioned earlier [35,42], and have a working prototype compiler for ParAlfl that generates serial combinators for direct execution on Alfalfa.

5.2 The ParLance Programming Environment

Another important issue, of course, is the programming environment that we provide to the user. Most existing environments on parallel machines are at a very low level of sophistication, and much research is needed to determine what kinds of development and debugging tools are appropriate. Yet we believe that the para-functional programming paradigm has some definite advantages over more conventional approaches when it comes to designing suitable programming environments:

- First, because there is a lack of side-effects in the language, one does not have to worry about timing issues in debugging the *functional* aspects of a program. In fact, one can do such debugging entirely on a sequential machine, where the annotations are simply ignored. Then, with the minor constraints on the annotations mentioned earlier, one is *guaranteed* that the same program will run "correctly" on a parallel machine (i.e., it will return the same answer).

- Second, because we have a formal, deterministic semantics for, as an example, the mapping annotations, we can easily build an interpreter that mimics this behavior. (Indeed, a prototype interpreter for the mapping annotations has already been built [33].) So to some extent even operational semantics may be debugged independently of parallel hardware. We envision a graphical interface as being the ideal mechanism for conveying such information; perhaps even a real-time display of the computation as it "unfolds."

- Finally, if the timing and mapping annotations are derived formally and automatically as described in Section 4.2, then the user is guaranteed that the program will meet the performance criterion set forth in the source program.

Given these comments, what is left to debug on the parallel machine? If the implementation is faithful to the formal semantics (and we have to allow the user to at least

assume that!), then all that is left are certain concerns about timing and mapping that relate entirely to efficiency, not functionality. If the resulting performance (which can be monitored in straightforward ways) is not good enough, one is free to change the algorithm or performance criterion, or perhaps add annotations of one's own, and reassess the resulting performance. The difficulty here is relating performance to the modifications being made – for example, determining where the communication bottlenecks are, or in general finding imbalances in the distribution of work. We feel that this problem can be suitably solved by providing effective monitoring tools of the dynamic execution of the program. It is at least a far more tractable situation than the many problems that must be solved in more conventional approaches to parallel programming.

6 Remaining Work

Obviously, there remains musch work to be done. We view our current work as a symbiotic effort in which advances are made both in transformation strategies and in para-functional programming language design and implementation. The many issues that we wish to investigate include: the formal semantics of synchronizing expressions, issues involving language syntax, re-use (in a software engineeriing sense) of annotations, optimization of processor communication, strictness analysis on non-flat domains, treatment of architectural constraints during optimization, development of index expansion techniques, and development of condensation strategies.

Perhaps one of the most important issues for us to resolve is how the user "transitions" between programming with "explicit" annotations (those generated by the user) and "implicit" annotations (those generated automatically by the compiler). There is a clear advantage to allowing the user to combine both techniques within the same program, as well as combining both forms of automation, one for affine recurrences, the other for serial combinators. How does one coordinate these different compilation technques? Furthermore, are there degrees of regularity in programs that lie somewhere between affine recurrences and completely unconstrained functional programs? Although we do not yet have satisfactory answers to these questions, there are several partial solutions worth mentioning.

First of all, it is likely that language features (perhaps just keywords) will be useful when combining compilation techniques. For example, consider these two combined compilation situations:

1. One may wish to *partition* the multiprocessor, mapping (automatically) a set of affine recurrences to one partition, and mapping (manually) an unconstrained subprogram to some other.

2. The function bodies of a set of affine recurrences may be arbitrarily complex. Thus one may wish to use either explicit annotations or the serial combinator techniques to compile the function bodies. This may in turn imply the use of even more complex partitioning strategies, such as *clustering* of processors for local (corresponding to function body) computations.

The first example demonstrates a "flat" partitioning strategy, the second a "nested" partitioning. Language features that indicate where affine recurrences begin and end, and that declare global partitioning strategies, may be useful in these situations.

Secondly, the compiler is likely to be *interactive*, both querying the user about design information and providing information about its decisions. As described earlier, the intermediate output of the compiler is just a (perhaps heavily) annotated version of the source program. It is here that the annotations can be seen *in toto*, and the user is free to examine, and possibliy modify, them.

Finally, we plan to use a structured editor to provide alternative "views" of an annotated program. In particular, the editor will be able to display the program with all annotations, with no annotations, or with certain classes of annotations chosen by the user. If a color display is available, the annotations can be highlighted in the obvious way.

7 Conclusions

We have described the foundation of a versatile parallel programming environment called *ParLance* that is based on a functional programming paradigm augmented with annotations for refinement of parallel operational behavior, sophisticated compiler technology for automatic decomposition and scheduling, and software development tools tailored to parallel computing. ParLance is firmly based on mathematical principles, making it possible for one to systematically develop parallel programs with predictable, quantifiable performance.

Using ParLance a programmer can optionally (1) write programs with no concern whatsoever for parallelism, (2) annotate programs in ways that are likely to improve performance, or (3) write programs that satisfy certain semantic constraints (those of affine recurrence equations) and automatically compile them efficiently for a particular machine. In all cases the same base language is used, and in fact programs in all three categories can be mixed freely, and could easily be combined into a single large application.

There are several aspects of our methodology that we think significantly facilitate program development. First, the *functional* aspects of a program are effectively separated from most of the *operational* aspects. This is manifested both in our compilation strategies and in the use of annotations to express operational behavior. By making the separation

visible in the language, one can isolate, reason about, and thus debug, the two classes of problems more-or-less independently.

Second, the multiprocessor is viewed as a *single autonomous computer* onto which a program is mapped, rather than as a group of independent processors requiring complex communication and synchronization. There are no special "parallelizing" constructs, no message-passing constructs, and in general no forms of "excess baggage" to express the rather simple notions of "where and when to compute things."

Third, sophisticated compiler technology is used to generate very efficient programs (indeed, optimal programs, if given enough processors) with respect to certain performance criteria, for a reasonably broad class of problems. For certain compute-intensive applications this programming option is essential.

Finally, the annotations used to refine the operational behavior are natural, concise, and have the property (with some minor constraints) that if a program is stripped of its annotations, it is still a perfectly valid functional program. A program may be written and debugged on a uniprocessor that ignores the annotations, and then executed on a parallel processor for increased performance. Portability is enhanced, since only the annotations need to change when one moves from one parallel topology to another (unless the algorithm itself changes). The ability to debug a program independently of the parallel machinery is invaluable.

References

[1] H.M. Ahmed, J.-M. Delosme, and M. Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *IEEE Computer*, 15:65–82, 1982.

[2] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M.S. Lam, O. Menzilcioglu, K. Sarocky, and J.A. Webb. Warp architecture and implementation. *IEEE Computer*, :346–56, 1986.

[3] E. Arnould, H.T. Kung, O. Menzilcioglu, and K. Sarocky. A systolic array computer. In *Proc. IEEE ASSP*, pages 232–235, 1985.

[4] J.L. Baer. *Computer Systems Architecture.* Computer Science Press, 1980.

[5] H.G. Baker and C. Hewitt. *The incremental garbage collection of processes.* AI Working Paper 149, Mass. Institute of Technology, July 1977.

[6] J.M. Barth. Shifting garbage collection overhead to compile time. *CACM*, 20(7):513–518, 1977.

[7] A. Bloss and P. Hudak. Variations on strictness analysis. In *Proc. 1986 ACM Conf. on LISP and Functional Prog.*, pages 132–142, ACM, August 1986.

[8] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1985.

[9] F.W. Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Trans. on Prog. Lang. and Sys.*, 6(2), April 1984.

[10] F.W. Burton. Controlling speculative computation in a parallel functional language. In *Int'l Conf. on Distributed Computing Systems*, pages 453–458, May 1985.

[11] McDowell C.E.. and W.F. Appelbe. Processor scheduling for linearly connected parallel processors. *IEEE Trans. Comp.*, C-35:632–638, 1986.

[12] M.C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Sym. on Prin. of Prog. Lang.*, pages 238–252, ACM, 1977.

[14] J.-M. Delosme and I.C.F. Ipsen. Design methodology for systolic arrays. In *Proc. SPIE Symp. 696*, 1986.

[15] J.-M. Delosme and I.C.F. Ipsen. Efficient systolic arrays for the solution of Toeplitz systems: an illustration of a methodology for the construction of systolic architectures in VLSI. In *Systolic Arrays*, Adam Hilger, 1986. To appear.

[16] J.-M. Delosme and I.C.F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in VLSI. In *Proc. Second Int. Symposium on VLSI Technology, Systems and Applications*, pages 268–273, Taipei, Taiwan, 1985.

[17] J.-M. Delosme and I.C.F. Ipsen. Parallel solution of symmetric positive definite systems with hyperbolic rotations. *Linear Algebra and its Applications*, 77:75–111, 1986.

[18] J.-M. Delosme and I.C.F. Ipsen. Systolic array synthesis: computability and time cones. In *Algorithmes et Architectures Paralleles*, North Holland, 1986.

[19] D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn. A second opinion on data flow machines and languages. *Computer*, 15(2):58–69, February 1982.

[20] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, Department of Computer Science, expected Spring 1987.

[21] B. Goldberg and P. Hudak. Alfalfa: distributed graph reduction on a hypercube multiprocessor. In *Proceedings of the Santa Fe Graph Reduction Workshop*, page to appear, Los Alamos/MCC, Springer-Verlag LNCS ..., October 1986.

[22] B. Goldberg and P. Hudak. *Inferring sharing properties of partial applications in higher-order functional languages.* Research Report in preparation, Yale University, Department of Computer Science, November 1986.

[23] G.H. Golub and C.F. van Loan. *Matrix Computations.* The Johns Hopkins Press, Baltimore, MD, 1983.

[24] D.H. Grit and R.L. Page. Deleting irrelevant tasks in an expression oriented multiprocessor system. *ACM Transactions on Programming Languages and Systems*, 3(1):49–59, January 1981.

[25] A.N. Haberman. *Path Expressions.* Technical Report, Carnegie-Mellon University, June 1975.

[26] D.E. Heller and I.C.F Ipsen. Systolic networks for orthogonal decompositions. *SIAM J. Sci. Stat. Comp.*, 4:261–269, 1983.

[27] D.E. Heller and I.C.F Ipsen. Systolic networks for orthogonal equivalence transformations and their applications. In *Proc. Conference on Advanced Research in VLSI, 1982*, pages 113–122, Artech House, Inc., 1982.

[28] P. Henderson. Functional programming, formal spepcification, and rapid prototyping. *IEEE Trans. on SW Engineering*, SE-12(2):241–250, 1986.

[29] W.D. Hillis. *The Connection Machine.* The MIT Press, 1985.

[30] C.-H. Huang and C. Lengauer. Mechanically derived systolic solutions to the algebraic path problem. In *CompEuro '87*, page , 1987.

[31] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[32] P. Hudak. *Collecting interpretations of expressions.* Research Report YALEU/DCS/RR-497, Yale University, Department of Computer Science, 1986.

[33] P. Hudak. Denotational semantics of a para-functional programming language. *Int'l Journal of Parallel Programming*, 15(2), 1986.

[34] P. Hudak. Distributed task and memory management. In *Proc. of Symposium on Principles of Distributed Computing*, pages 277–289, ACM, August 1983.

[35] P. Hudak. Para-functional programming. *Computer*, 19(8):60–71, August 1986.

[36] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proc. 1986 ACM Conf. on LISP and Functional Prog.*, pages 351–363, ACM, August 1986.

[37] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 300–314, ACM, 1985.

[38] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. In *Proceedings of 1985 Int'l Conf. on Parallel Proc.*, pages 831–839, August 1985. Also appeared in IEEE Trans. on Computers, Vol C-34, No. 10, October 1985, pages 881-891.

[39] P. Hudak and B. Goldberg. Experiments in diffused combinator reduction. In *Proc. 1984 ACM Conf. on LISP and Functional Prog.*, pages 167–176, ACM, August 1984.

[40] P. Hudak and B. Goldberg. Serial combinators: "optimal" grains of parallelism. In *Functional Programming Languages and Computer Architecture*, pages 382–388, Springer-Verlag LNCS 201, September 1985.

[41] P. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proc. 1982 ACM Conf. on LISP and Functional Prog.*, pages 168–178, ACM, August 1982.

[42] P. Hudak and L. Smith. Para-functional programming: a paradigm for programming multiprocessor systems. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 243–254, January 1986.

[43] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 97–109, January 1986.

[44] J. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, pages 129–146, Springer-Verlag LNCS 201, September 1985.

[45] T. Johnsson. *The G-machine: an abstract machine for graph reduction.* Technical Report, PMG, Dept. of Computer Science, Chalmers Univ. of Tech., February 1985.

[46] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th Sym. on Prin. of Prog. Lang.*, pages 296–306, ACM, January 1986.

[47] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J. Appl. Math*, 14:1390–1411, 1966.

[48] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14:563–590, 1967.

[49] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. *IEEE Computer*, 17(7):70–82, July 1984.

[50] D.E. Knuth. An empirical study of fortran programs. *Software and Experience*, 1:105–33, 1971.

[51] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[52] H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282, SIAM, Philadelphia, PA, 1978.

[53] M.S. Lam and J. Mostow. A transformational model of VLSI systolic design. *IEEE Computer*, :42–52, 1985.

[54] C.E. Leiserson. *Area-Efficient VLSI Computation.* The MIT Press, 1983.

[55] B.C. Levy. *2-D Polynomial and Rational Matrices, and their Applications for the Modeling of 2-D Dynamical Systems.* PhD thesis, Dept of Electrical Engineering, Stanford University, 1981.

[56] C. Lutz, S. Rabin, C. Seitz, and D. Speck. Design of the MOSAIC element. In *Proc. Conference on Advanced Research in VLSI, 1984*, pages 1–10, Artech House, Inc., 1984.

[57] D.I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. Comp.*, C-31:1121–1126, 1982.

[58] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comp.*, C-35:1–12, 1986.

[59] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[60] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. Comp.*, CC-29:763–776, 1980.

[61] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, 29:1184–201, 1986.

[62] S. Pappert. *Mindstorms: Children, Computers and Powerful Ideas.* Basic Books, 1980.

[63] V. Pratt. Modelling concurrency with partial orders. *Int'l Journal of Parallel Programming*, 15(1):33–72, February 1986.

[64] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual Intern. Symp. Computer Architecture*, pages 208–214, IEEE, 1984.

[65] S.K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Dept of Electrical Engineering, Stanford University, 1985.

[66] Y. Saad. Gaussian elimination on hypercubes. In *Algorithmes et Architectures Paralleles*, North Holland, 1986. to appear.

[67] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Trans. on Prog. Lang. and Systems*, 7(2):299–310, 1985.

[68] R. Schreiber. Systolic arrays for eigenvalue computation. In *Proc. SPIE 341 (Real Time Signal Processing V)*, pages 27–34, 1982.

[69] J. Schwarz. Verifying the safe use of destructive operations in applicative programs. In B. Robinet, editor, *Program Transformations – Proc. of the 3rd Int'l Sym. on Programming*, pages 395–411, Dunod Informatique, 1978.

[70] C.L. Seitz. The cosmic cube. *CACM*, 28:22–33, 1985.

[71] E. Shapiro. Systolic programming: a paradigm of parallel processing. In *Proc. Int. Conf. Fifth-Generation Computer Systems*, pages 458–470, 1984.

[72] E. Shapiro. *Systolic programming: a paradigm of parallel processing.* Dept. of Applied Mathematics CS84-21, The Weizmann Institute of Science, August 1984.

[73] N.S. Sridharan. *Semi-applicative programming: an example.* Technical Report, BBN Laboratories, November 1985.

[74] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.