Para-Functional Programming:
A Paradigm for Programming Multiprocessor Systems

Paul Hudak and Lauren Smith

Research Report YALEU/DCS/RR-390
January 1985 (revised June 1985)

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520
Arpanet: hudak@yale, smith-lauren@yale

This research was supported in part by NSF Grant MCS-8302018,
and a Faculty Development Award from IBM.

Table of Contents

1. Introduction 1
2. ALFL: A Simple Functional Language 4
3. ParAlfl = ALFL + Annotations 5
    3.1. Mapped Expressions 5
    3.2. Eager Expressions 8
    3.3. Notes on Determinacy 10
4. Sample Application Programs 11
    4.1. Parallel Factorial 11
    4.2. A Prime Number Generator 13
    4.3. Matrix-Vector Product 15
    4.4. Solution to Upper Triangular Block Matrix 18
5. A Formal Semantics for ParAlfl 20
    5.1. Standard Interpretation 20
    5.2. Determinacy Theorem Revisited 21
    5.3. Execution Trees 22
    5.4. Semantics of Mapped Expressions 22
6. Conclusions and Future Work 24
7. Acknowlegements 25

List of Figures

Figure 3-1: Two possible network topologies 6
Figure 4-1: Annotated divide-and-conquer program to compute k! 11
Figure 4-2: Dynamic flow of data for parallel factorial 12
Figure 4-3: Divide-and-conquer factorial with unique behavior at leaves 12
Figure 4-4: Sieve of Erastothenes on an "infinite" vector of processors 13
Figure 4-5: Parallel version of Sieve of Erastothenes 14
Figure 4-6: Sieve of Erastothenes using "piece-meal" parallelism 14
Figure 4-7: Matrix-vector product annotated for a ring topology 16
Figure 4-8: The construction of **blist** (**n**=4) 17
Figure 4-9: The computation of the result vector (**n**=4) 17
Figure 4-10: Algorithm for solving **Ux=b** on a ring 18
Figure 4-11: Program for solving **Ux=b**, annotated for a ring 18
Figure 4-12: Time sequence of program to solve **Ux=b** 19

Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems

Paul Hudak
Lauren Smith

Research Report YALEU/DCS/RR-390
January 1985 (revised June 1985)

Yale University
Department of Computer Science

Abstract

One of the most important pragmatic advantages of functional languages is that concurrency in a program is *implicit* – there is no need for special constructs to express parallelism as is required in most conventional languages. Furthermore, it is fairly easy for systems to automatically determine the concurrency and thus decompose a program for execution on a suitable parallel architecture. Yet it is often the case that one knows *precisely* the *optimal* decomposition for execution on a particular machine, but one can never expect a compiler to determine such optimal mappings in all cases. This paper is concerned with ways to allow the programmer to *explicitly* express this mapping of program to machine, by using annotations that, given a few minor constraints, cannot alter the functional semantics of the program. We show through several detailed examples the power and conciseness of the resulting "para-functional" programming methodology, using an experimental language called *ParAlfl* based on our ideas. We also give a formal denotational description of the mapping semantics using a notion of *execution trees*.

This research was supported in part by NSF Grant MCS-8302018.

1. Introduction

The advantages of functional languages have been well-argued by functional programming advocates in the past several years. One of the most important *pragmatic* advantages is that functional programs expose parallelism in a "natural way," and that it is easy for a compiler to detect such parallelism for exploitation on a suitable parallel architecture. The lack of side-effects accounts (at least partially) for the well-known Church-Rosser Property [31] that guarantees determinacy in the resulting parallel computation. Indeed, many of the earlier functional languages were developed simultaneously with work on dataflow or reduction machines (for example, VAL and the Static Dataflow Machine [8, 25], ID and the U-interpreter [1, 2], DDN and the dataflow machine DDM1 [6, 7], and FGL and the reduction machine AMPS [20, 21]). In all of these efforts the parallelism is detected *automatically* by the system – the user in no way has to provide extraneous information such as that needed in most imperative programming languages designed for parallel computation.

Recently there has also been a great deal of interest in so-called "network computers" or "ensemble architectures," characterized as a collection of autonomous processing elements with only local store, interconnected by a homogeneous communications network, and communicating by "messages." The interest in this style of machine is actually not surprising, for many reasons. Not only do they avoid the classical "von Neumann bottleneck" by being effectively decentralized, but they are also extensible, and in general quite easy to build. Indeed several existing machines meet this description, such as the Butterfly Multiprocessor [4], Cosmic Cube [28], Intel iPSC [19], and ZMOB [], to name a few, and there are many proposed machines whose construction is not complete.

The combination of functional languages and network computers thus seems like a natural one. Indeed, considerable research is underway in this direction, such as Keller's Rediflow Multiprocessor [23] and the author's work on DAPS [14, 16, 17]. Such work shows considerable promise: a functional program is automatically decomposed for dynamic distribution on a network of processors, typically by some sort of "load-balancing" or "diffusion-scheduling" strategy, and execution takes place as the processors cooperatively accomplish a global form of graph-reduction. Simulated performance figures are quite encouraging, and it is hoped that the resulting systems will perform quite well on a wide variety of programs.

We prefer to view the aforementioned systems as *general-purpose multi-user computer systems*. Yet it is often the case that one has a *dedicated* parallel machine for a particular application, and furthermore that one knows *precisely* the *optimal mapping* of one's program onto that machine. One can never expect an automated system to determine this optimal mapping in all cases (indeed, in the general case such a task is undecideable), so it is desireable to allow the user 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 architectures. As it stands, there are almost no languages providing this capability.

Our goal then, is to remedy this situation. Since functional languages seem generally well-suited to parallel computing, we use it as a basis for the following simple solution: a functional program, being essentially an immutable object, may be mapped to a machine by *annotating* its subexpressions, in such a way that the program's functional behavior is not altered; i.e., the program itself remains unchanged. We refer to the resulting methodology as *para-functional programming,* since not only does it provide a much-needed tool for expressing parallel computation, but it also provides an operational semantics that is truly "extra," or "beyond," the functional semantics of the program. The resulting methodology is quite powerful, for several reasons:

First, *it is very flexible.* Not only is the idea easily adapted to any functional language, but also any network topology may be captured by the notation, since no a priori assumptions are made with regard to the logical structure of the physical system. All of the benefits of conventional scoping disciplines are available to create modular programs that conform to the topology of a given physical system.

Second, *the annotations are natural and concise.* There are no special control constructs, no message-passing constructs, and in general no forms of "excess baggage" to express the rather simple notion of "where and when to compute things." We will show through several non-trivial examples the perspicuous nature of the annotations, and that very few annotations are required to express most typical mappings.

Finally, with some minor syntactic constraints, *if a para-functional program is stripped of its annotations, it is still a perfectly valid functional program.* This means that it can be written and debugged on a uniprocessor without the annotations, and then executed on a parallel system by adding the annotations 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.

Relationship to Other Work

In spirit, the work that is most similar to ours is Shapiro's "systolic programming" in Concurrent Prolog [29], which is in turn derived from earlier work on "turtle programs" in Logo [27]. However, there are important differences. First, both of these earlier efforts have a notion of "directionality;" i.e., a notion of a process (or turtle) "navigating" through a network (or turtle world) by "facing" in a certain direction. Our approach is more general, in that such navigation is simply a special case of a particular mapping. We can "skip" from one processor to any other just as easily as to its neighbor. Another important difference, of course, is the programming paradigm on which the extensions are based. Logo is a conventional imperative language, and has all of the problems that one might expect with a language having side effects. Concurrent Prolog has the purity of a functional language, but is of course based on logic rather than functions.

It is interesting to note that in logic programming the success of these ideas seems to rely on the notion of "read-only variables" (as well as a "commit" operator) which are an unnecessary distinction in functional languages since in a sense *all* variables are read-only (of course, this can also be viewed as a disadvantage, since the general utility of unification is unavailable). Also, the conventional block scoping rules of functional languages seem to give one an added level of control over the use of variables, and thus over the movement of data through the network. In particular, free variables in a mapped expression may reference objects computed on any arbitrary processor. This added power slightly complicates the semantics, as does the use of higher-order functions, but we believe it is worthwhile. The formal semantics given in Section 5 is the first that we know of dealing with these issues.

Other related efforts include that of Keller and Lindstrom [24], who independently (in the context of functional databases) suggest the use of annotations very similar to our mapped expressions. Their purpose is to permit users to assign database objects to particular "sites" in a distributed network, a very useful capability. We have generalized that work by widening the application domain, and providing an additional annotation to express "eager" evaluation. The latter annotation is similar to Burton's

annotations to the lambda calculus to control reduction order; in particular, to provide control over "lazy," "eager," and "parallel" execution [5]. Finally, we should point out that the "exposure" of the operational notion of "the currently executing processor" is inherently a process of "reflection" as used by Smith [30] and more recently by Friedman and Haynes [9].

Overview of Paper

In the next section we describe a functional language called ALFL that serves as a test-bed for our ideas. When extended with the annotations, we call the resulting para-functional programming language *ParAlfl*, and it is described in detail in Section 3. We then present several non-trivial examples in Section 4 that demonstrate the proposed programming paradigm. Having read through these examples the reader should have a good intuitive feel for the semantics of ParAlfl, which we strengthen in Section 5 with a formal denotational semantics. In Section 6 we summarize our work and discuss future research directions.

2. ALFL: A Simple Functional Language

ALFL [15] is a block-structured, lexically-scoped functional language with lazy evaluation semantics, similar in style to FEL [22] and SASL [32]. We describe it only briefly here, with the assumption that the reader is familiar enough with this style of language that the examples will be mostly self-explanatory. The saliant features of ALFL are:

- A program is an *equation group*, having the following form:

```
{ f1 x1 x2 ... == e1;
  f2 y1 y2 ... == e2;
  ...
  $result exp;
  ...
  fn z1 z2 ... == en }
```

  An equation group is delimited by curly brackets, contains a collection of *equations* that define local identifiers (**f1** though **fn**), and has a single *result clause* that expresses the value to which the equation-group will evaluate. (For clarity reserved words such as **$result** are always prefixed with $.) Equation groups are just expressions, and may thus be nested to an arbitrary depth.

- Equations are mutually-recursive, are evaluated "by demand," and thus their order is irrelevant. A double equal-sign ("==") is used in equations to distinguish it from the infix operator for equality.

- Functions defined by equations are "curried," and function application associates to the left. Thus **f x y == body** defines the function **f** that takes one argument and returns another function that takes one argument, finally returning **body**. Definitions of simple values, such as **x == exp**, can be viewed simply as nullary functions.

- A conditional expression has the form "**pred -> cons, alt**" and is equivalent to the more conventional "**if pred then cons else alt.**"

- Lists are constructed "lazily." The symbols "^" and "^^" are infix operators for cons and append, respectively, and **hd** ("head") and **tl** ("tail") are like car and cdr, respectively, in Lisp. A proper list may also be constructed using brackets, as in **[a,b,c]** (which is equivalent to **a^b^c^[]**).

- ALFL has a *pattern-matcher* through which complex functions may be defined more easily. For example, the function **member** may be defined by:

```
member x []  = false;
   '   x (x^L) = true;
   '   x (y^L) = member x L;
```

  When using the pattern-matcher note that the order of equations defining the same function *does* matter. Also note the use of a single quote ' as a shorthand for the function name in consecutive equations defining the same function.

- ALFL also has functional vectors and arrays. The primitive function call **mkv n f** creates a vector **v** of **n** values, indexed from **0** to **n-1**, such that its **i**th element **v i** is the same as **f i**. Similarly, the expression **mka [d1,d2,...,dn] f** returns an **n**-dimensional array **a** such that **a x1 ... xn = f x1 ... xn**. Note that vectors and arrays defined in this way are used syntactically just like functions.[1]

There are other syntactic and semantic features of ALFL, but they are beyond the scope of this paper (the interested reader should refer to [15] for more details). Our purpose here is primarily to give a framework on which to build the extensions that make up ParAlfl.

3. ParAlfl = ALFL + Annotations

ParAlfl is an experimental para-functional programming language based on ALFL. The extensions that we propose, however, could easily be added to almost any functional language. In this section we present the syntax and an intuitive operational semantics for the extensions, of which there are essentially only two: *mapped expressions* and *eager expressions*. We return to a formal denotational description of ParAlfl in Section 5.

3.1. Mapped Expressions

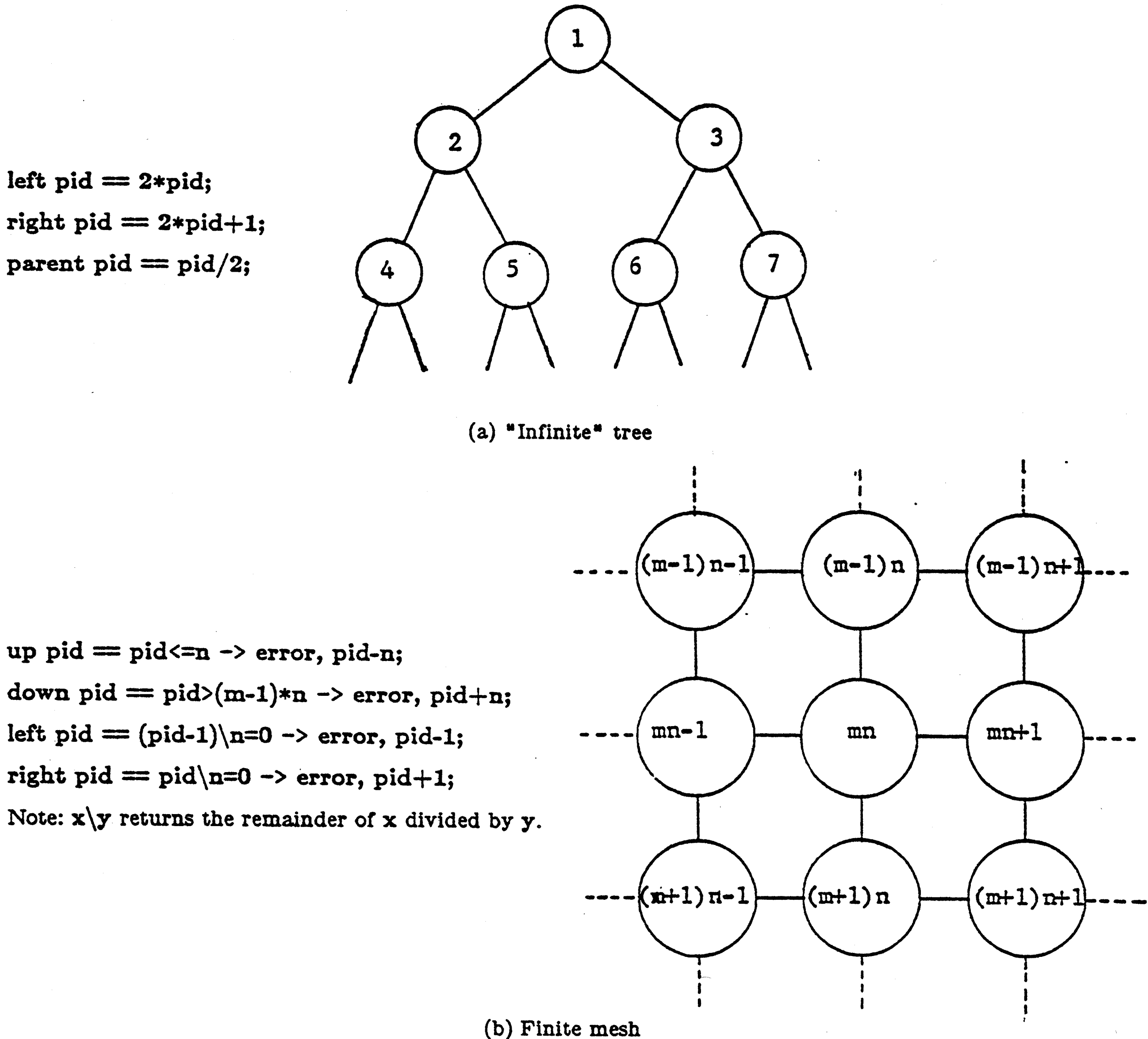
As mentioned in the Introduction, our primary goal is to allow one to map the evaluation of a program onto any particular network topology. We accomplish this by using *mapped expressions*, which have the simple form:

exp $on proc

which intuitively declares that **exp** is to be computed on the processor identified by **proc**. The expression **exp** is the *body* of the mapped expression, and represents the value to which the overall expression will evaluate (and thus can be any valid ParAlfl expression, including another mapped expression). The expression **proc** must evaluate to a processor id, or *pid*. We will assume, without loss of generality, that processor ids are *integers*, and that there is some pre-defined mapping from those integers to the physical

[1] Indeed, the function mkv can be viewed simply as a caching functional!

processors they denote. For example, a tree of processors might be numbered as shown in Figure 3-1a, or a mesh as shown in Figure 3-1b. The advantage of using integers is that the user may manipulate them using conventional ALFL machinery; for example Figure 3-1 also defines functions that map pids to neighboring pids.[2]



(b) Finite mesh

Figure 3-1: Two possible network topologies

To facilitate the use of mapped expressions, we also introduce a way for the user to access the *currently executing processor*. This is a *dynamic* notion, since a recursive expression, for example, might be evaluated on different processors, depending on the depth of the recursion. We provide this access through the reserved identifier $self which when evaluated returns the pid of the currently executing

---

[2] A safer discipline might be to define a pid as a unique type, and to provide primitive functions to manipulate values having that type.

processor. It should be pointed out that since **$self** is essentially a "dynamic variable," its evaluation violates the normal notion of *referential transparency* since it will have different values depending on where and when in the program it is evaluated! This seemingly unfortunate state of affairs, however, may be remedied by imposing the (reasonable) syntactic constraint that the identifier **$self** cannot appear other than in the pid expression of a mapped expression. Since the value of a mapped expression **exp $on pid** is **exp**, then with the additional constraint that all pid expressions must terminate, it is easy to see that *the value of a program cannot change, no matter how many expressions are mapped.* This simple determinacy property is a very important one, since it allows a program to be written and debugged on a uniprocessor without the annotations, and then executed on a parallel system by adding the annotations for better performance.

It should also be noted that the information provided by **$self** is philosophically no different from providing the depth of the current execution stack, or the value of the current program counter, or the register containing the value of a certain variable, or any other arbitrary implementation-dependent parameter. It's just that the currently executing processor can be used to great advantage, as the examples in the next section will demonstrate. From a semantic viewpoint, just as the meaning of an expression is normally given as a function of a "current continuation" and "current environment," the operational meaning that we are trying to convey is given as a function of a "currently executing processor." And just as in Scheme one is given access to the current continuation via calls to the primitive function **call/cc**, we are providing access to the currently executing processor via the dynamic variable **$self**.

Simple Examples of Mapped Expressions

As a simple example of the use of mapped expressions, consider the program fragment **f(x) + g(y)**. The semantics of the + operator allows the two subexpressions to be executed in parallel. If we wish to express precisely *where* the subcomputations are to occur, we may do so by annotating the expression, as in:

```
(f x $on 0) + (g y $on 1)
```

where **0** and **1** are processor ids. Of course, this static mapping is not very interesting, but by using the dynamic variable **$self** we can be more creative. For example, suppose we have a mesh or tree of processors such as in Figure 3-1. We may then write:

```
(f x $on left $self) + (g y $on right $self)
```

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

Similar mappings can be made from composite objects such as vectors and arrays to specific multiprocessor configurations. For example, if **f** is defined by **f i == i**2 $on i**, then the call **mkv n f** will produce a vector of squares, one on each of the processors, such that the **i**th processor contains the **i**th element (namely i^2). Further suppose we have two vectors **v** and **w** (which may be distributed as

above, but are not required to be), and we wish to create a third being the sum of the other two, but distributed over the **n** processors. This can be done very simply by: **mkv n g** where **g i == (v i + w i) $on i.**

A Comment on Lexical Scoping and Data Movement

It is important to note that *no* communications primitives and *no* special synchronization constructs are needed in the above examples to "move data" from one processor to another. Together with the annotations, it is accomplished simply through the use of normal lexical scoping mechanisms. For example, consider the program fragment:

```
{ x == exp $on p;
  $result ...
           (...x...) $on q
           ...
           (...x...) $on r
}
```

The value of **x** is created on processor **p**, and the two references to **x** essentially cause that value to be "sent" to processors **q** and **r**. Since there are no side-effects, there is no need to synchronize the concurrent accesses to **x**'s value. If "finer" control over **x**'s movement is desired, intervening references can be made, as in:

```
{ x == exp $on p;
  $result ...
           { x == x $on s;
             $result ...
                      (...x...) $on q
                      ...
                      (...x...) $on r }
}
```

which essentially moves **x** to processor **s** prior to its access from processors **q** and **r**.

We feel that the environments created through lexical scoping fit naturally into the parallel computing world, an idea supported by all of the examples given in Section 4. It is a simple yet powerful way to express communication in a parallel computer, and eliminates the need for special message-passing constructs as is used in almost every other language for parallel computing.

3.2. Eager Expressions

The second form of annotation arises out of the occasional need for the programmer to "override" the lazy evaluation strategy of ALFL, which normally does not evaluate an expression until it is absolutely needed. Opportunities to override lazy evaluation can often be inferred by a suitable "strictness analysis" [18, 26], but there are cases where such an analysis will fail, and in any case the programmer may wish to make such inferences explicit in the source program. We thus introduce an *eager expression*, which has the simple form:

#exp

and intuitively forces the evaluation of **exp** *in parallel* with its "most relevant" surrounding syntactic form, as defined by:

if **#exp** appears as:	it executes in parallel with:	for example:
argument in function call	the function call	**f x #y z**
element of list	the list	**[x, #y, z]**
arm of conditional	the conditional	**p -> #x, y**
operand of infix operator	the whole operation	**x^#y, x&#y**

Thus, for example, in the expression **p -> f #x y, z**, the evaluation of **x** begins as soon as **p** has been determined to be **true**, and simultaneously the function **f** is invoked on its two arguments. Note that the evaluation of *some* subexpression begins when *any* expression is evaluated, and thus to eagerly evaluate that subexpression accomplishes nothing. For example, note the following equivalences:

#p -> x, y ≡ p -> x, y
#x & y ≡ x & y
#f x y ≡ f x y

Similarly, there is no need to eagerly evaluate either argument to a strict binary operator such as +, since in ParAlfl it is assumed that both arguments may be evaluated in parallel.

The *value* of an expression containing an eager subexpression is the same as if the expression had no annotation at all. Thus, as with mapped expressions, the annotation only adds an *operational* semantics, and means that the user may invoke a non-terminating computation yet have the overall program terminate. For example, consider:

```
{ f x == 1;
  g y == g (y+1);
  $result f #(g 0) }
```

The call to **g** will be invoked eagerly and will not terminate, but since **f** does not depend on the value of its argument, the program will terminate and return the value **1**. The process that is computing **g 0** is often called an *irrelevant task* (once its value has been determined to be no longer needed). There exist strategies for finding and deleting irrelevant tasks at run-time [3, 11, 12, 13], but such mechanisms are well beyond the scope of this paper. Suffice it to say that given such a mechanism there are real situations where one might wish to exercise the option of invoking a non-terminating computation. An example of this is given in Section 4.2.

Simple Examples of Eager Expressions

Together with the conditional, we can define a simple function to evaluate two expressions in parallel:

par a b == false -> #a, #b

Thus **par a b** will return the value **b**, but **a** is eagerly evaluated in parallel – note that **par a b** terminates even if **a** doesn't. This function is useful in eagerly starting the evaluation of a subexpression that the programmer either *knows* will eventually be needed, or just *conjectures* will eventually be needed and is willing to allocate the resources to compute it. Another use is given in the example below.

To force the computation of all elements of a "lazy" list, we can use the following function:

```
strong-force l == { $result sf l;
                    sf  []   == l;
                    sf (a^lst) == par a (sf lst) }
```

This function actually evaluates *each element* of the list. If all we were interested in was *expanding* the list, but not computing the individual elements, we could define the "less forceful" function:

```
weak-force l == { $result wf l;
                  wf  []   == l;
                  wf (a^lst) == wf lst }
```

which uses no annotations at all! Unfortunately, both of these functions tend to "strictify" lists; that is, the end of the list must be reached before anything is returned, meaning that the function will not terminate when applied to an infinite list. This can be remedied by changing the result clause in the functions **strong-force** and **weak-force** to: **$result par (sf l) l** and **$result par (wf l) l**, respectively.

3.3. Notes on Determinacy

All ParAlfl programs possess the following determinacy property, which we state as a theorem:

> **Theorem 1:** (Informal) A ParAlfl 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 annotations removed. That is, both programs return the same value.

The reason for the first constraint was discussed earlier: **$self** can return different values depending on the mapping strategy used. The purpose of the second constraint should be obvious: if the system diverges when determining on which processor to execute the body of a mapped expression, then it will never get around to computing the value of that expression. We postpone a formal proof of this theorem until a formal denotational semantics for ParAlfl is given in Section 5, at which point the proof becomes trivial.

Although neither determinacy constraint is severe, there are practical reasons for wanting to violate the first one; i.e., for wanting to use the value of **$self** in other than a pid expression. The most typical situation where this arises is in a non-isotropic topology where certain processors form a "boundary" for the network. For example, the leaf processors in a tree, or the edge processors in a mesh. There are many distributed algorithms whose behavior at such boundaries is different from their behavior at internal nodes. To express this, one needs to know when execution is occurring at the boundary of the network, which can be conveniently determined by analyzing the value of **$self**. An example of this is given in Section 4.1.

One final note on determinacy concerns the use of #. Although this annotation does not affect the program's functional behavior, there may be situations where one wishes to make an expression "strict" in one of its subexpressions. Unfortunately this can prevent a program from terminating when otherwise it would, thus changing its functional semantics. Our approach to providing this capability is therefore not through annotations, but through a primitive function that induces the strictness property. The

predicate **terminate** is defined such that **terminate exp** returns **true** if **exp** terminates; otherwise it does not terminate either. We can then define:

strictify e1 e2 == terminate e1 -> e2, e2

so that, for example, **strictify x (f x y z)** will essentially make the call to **f** strict in **x** (and, as an aside, will cause the evaluation of **x** *before* that of **f x y z**). This definition depends on the fact that in both ALFL and ParAlfl, ($\perp$ -> **x,x**) = $\perp$ (where $\perp$ is the symbol used in semantics to represent the value of a non-terminating computation).[3]

4. Sample Application Programs

It has been our experience in looking at many multiprocessing algorithms that they are quite functional in their global behavior, perhaps because their parallel nature precludes dependencies on a centralized shared memory, and thus they fall nicely into the functional model. In this section we present four non-trivial examples that demonstrate this, showing the power and flexibility of para-functional programming. We urge the reader to note how few annotations are needed to accomplish the desired mappings.

4.1. Parallel Factorial

```
{ $result pfac 1 k $on root;

  pfac lo hi == lo=hi -> lo,
                lo=(hi-1) -> lo*hi,
                { $result (pfac lo mid $on left $self) *
                          (pfac (mid+1) hi $on right $self);
                  mid == (lo+hi)/2 };

  left  pe == (2*pe > n) -> pe, 2*pe;
  right pe == (2*pe+1 > n) -> pe, 2*pe+1;
  root == 1;
}
```

Figure 4-1: Annotated divide-and-conquer program to compute k!

Figure 4-1 shows a simple "parallel factorial" program, annotated for execution on a finite binary tree of $n = 2^d$ processors.[4] At each iteration, the computation is split into two parts and mapped onto the two "children" of the current processor, enabling parallel computation of the two subexpressions of the

---

[3] It may seem as if terminate need not be primitive if at least one other type-predicate is primitive. For example, suppose there is a predicate integer defined in the normal way. Then it seems as if the predicate terminate could be defined by:

terminate exp == integer exp -> true, true

Semantically speaking, this version of terminate has the property that terminate x = $\perp$ if x=$\perp$, otherwise true. Although this is also true of the primitive version of terminate described earlier, it does not fully capture the intended semantics, because with such a definition it might be possible for a compiler to infer that an expression will be an integer, thus constant-folding the overall expression to true! Our intent is more operational -- i.e., we want exp to actually be evaluated (if it hasn't been already), and return true only if it terminates. Thus terminate must be primitive so that the compiler can decide if certain transformations are applicable (in particular, it cannot reduce terminate exp to true even if it can infer that exp will terminate).

[4] Although this program computes factorial, almost any binary "divide and conquer" algorithm could fit into the same framework.

multiplication. Note that through the normal lexical scoping rules, **mid** will be computed on the "current" processor and passed to the child processors as needed (recall the discussion in Section 3.1). The functions **left**, **right**, and **root** describe the network mapping necessary for this topology. In this case, if the current processor is a leaf node, all further calls to **pfac** will be executed on that processor. In practice, more complex routing functions could be devised that, for example, might reflect the computation upward when a leaf processor was reached. Figure 4-2 shows the process mapping and the flow of data between processes for this example.

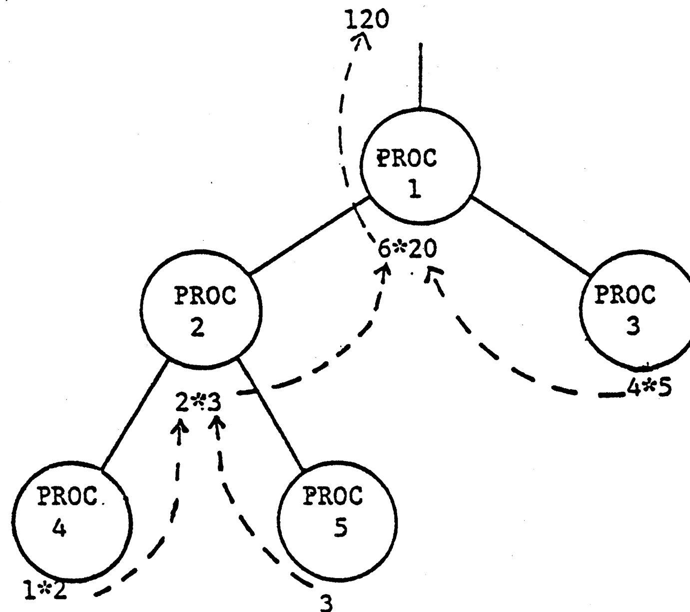


Figure 4-2: Dynamic flow of data for parallel factorial

```
{ $result pfac 1 k $on root;

  pfac lo hi == lo=hi -> lo,
                lo=(hi-1) -> lo*hi,
                { $result leaf?($self) -> sfac lo hi 1,
                          (pfac lo mid $on left $self) *
                          (pfac (mid+1) hi $on right $self);
                  mid == (lo+hi)/2 };

  sfac lo hi acc == lo=hi -> lo*acc,
                    sfac (lo+1) hi (lo*acc);

  leaf? pe == pe >= 2^(d-1)
  left  pe == 2*pe;
  right pe == 2*pe+1;
  root == 1;
}
```

Figure 4-3: Divide-and-conquer factorial with unique behavior at leaves

The above parallel factorial program observes the constraints stated in Theorem 1, and thus it is determinate regardless of the annotations. However, it may be desirable to use a more efficient factorial algorithm at the leaf nodes. An example of this is given in Figure 4-3, where the tail-recursive function **sfac** is invoked at the leaves. Note that even though **$self** is used in a non-pid expression, *the program is still determinate*. This is, of course, usually the case, but we cannot guarantee it in general without the previously discussed constraints.

4.2. A Prime Number Generator

Figure 4-4 shows a program to compute the first n prime numbers, using the well-known "sieve of Erastothenes." Ignoring the annotations for a moment, this program demonstrates a classical use of "infinite lists." First the infinite list of integers starting with 2 is generated (**ints**). The function **sift** then removes the first element, which it takes as prime, and elides multiples of that prime from the remaining list (by calling **filter**). This "filtered" list is then passed recursively to **sift** to form the rest of the infinite list of primes. The primitive function call **prefix n primes** then selects the first **n** elements from the result.

```
{ $result prefix n primes;

  primes == sift ints;

  sift (p^rest) == { $result p ^ (sift (filter rest) $on right $self);
                     filter (n^l) == n\p=0 -> filter l, n ^ filter l };

  numsfrom n == n ^ numsfrom (n+1);
  ints == numsfrom 2;
}
```

Figure 4-4: Sieve of Erastothenes on an "infinite" vector of processors

With the single mapping annotation shown, the subsequent calls to **sift** are mapped onto successive processors to the "right" of the current one. We will ignore the details of the function **right** that accomplishes this, recognizing that the conceptually infinite vector of processors could be a ring, "twisted torus," hypercube, or other topology. However, there is something terribly wrong with this program: It has no parallelism! Remember that lists are computed lazily, and thus there are no strict operators that could create parallelism, other than those in the trivial subexpression **n\p=0**.

To fix this problem, one should first note that the function **filter** is essentially doing all of the work -- it must check successive elements of **rest** until it finds one that is not a multiple of **p**. To get parallelism one needs to have several invocations of filter operating together, one "feeding" the next in an effectively pipelined manner. However, there is no simple way to call **filter** and have it return just enough elements to satisfy subsequent calls to **filter** so that exactly **n** primes are generated. The simplest solution is to make **filter** behave eagerly, extract the first **n** primes from the result eagerly, and let the system's task

manager "kill off" the then irrelevant processes computing the filtered streams. This solution is shown in Figure 4-5 -- note the eager call to **filter** and the redefinition of **prefix** to evaluate its elements eagerly (the other two calls to **filter** could also be made eagerly, but little additional parallelism would be attained, since they get evaluated "immediately" anyway).

```
{ $result prefix n primes;

  primes == sift ints;

  sift (p^rest) == { $result p ^ (sift (filter rest) $on right $self);
                     filter (n^l) == n\p=0 -> filter l, n ^ #(filter l) };

  numsfrom n == n ^ numsfrom (n+1);
  ints == numsfrom 2;

  prefix 0  l == [];
     '   n (a^l) == a ^ #(prefix (n-1) l)
}
```

Figure 4-5: Parallel version of Sieve of Erastothenes

As an aside, we should point out that this example demonstrates an important use of, and need for, an automatic task deletion mechanism as discussed in Section 3.2. In a "conventional" multiprocessing environment one might instead have a global variable that could be set after the **n** primes were extracted, which could serve as a signal for the **filter** processes to "kill themselves." This is not as clean a solution for at least two reasons: First, it requires the explicit coding of the termination condition into the function **filter**, and second, providing such explicit mechanisms introduces the possibility of programmer error, either terminating a process too soon or failing to terminate one that should. Providing an automatic task deletion mechanism in a parallel system is akin to providing automatic garbage collection in a sequential system -- it frees the programmer from the need to explicitly deallocate objects, whether they be processes or cons cells.

```
{ $result sift ints;

  sift (p^rest) == { new == filter rest;
                     $result par new (p ^ (sift new $on right $self));
                     filter (n^l) == n\p=0 -> filter l, n ^ filter l };

  numsfrom n == n ^ numsfrom (n+1);
  ints == numsfrom 2;
}
```

Figure 4-6: Sieve of Erastothenes using "piece-meal" parallelism

It is interesting to note that there is another useful parallel version of this program. Consider the following request from a client: create a lazy, infinite list, but every time an element is "demanded" from the list, return the element and simultaneously compute the next element in parallel so that it will be immediately available upon the *next* demand. Surprisingly, this seemingly complex request can be filled

simply by using **par** as defined earlier to eagerly evaluate **filter rest** when **sift** is called, thus computing the next non-multiple of **p** in advance. The resulting program is shown in Figure 4-6.[5] It is not hard to imagine generalizing this technique so that a "buffer" of n primes is maintained, ready to be "consumed" by some other process.

If nothing else, these examples of prime number generators amply demonstrate that parallelism in a program is not an "all-or-nothing" proposition. There are many subtle degrees of parallelism and mapping possibilities that the programmer may wish to express. In a conventional language augmented for parallel computing, expressing these alternatives typically involves changing the program's fundamental functional behavior. We believe that the functional and operational behaviors should be kept separate, and that the para-functional approach accomplishes this separation quite well. One can control not only the mapping of program to machine, but also the degree of parallelism manifested in the evaluation strategy, without jeopardizing the functional correctness of the program.

4.3. Matrix-Vector Product

The next two examples represent typical scientific computing applications, and are both annotated for execution on a *ring* of **n** processors, numbered **0** through **n-1**. Although the topology of a ring is rather simple, its limited interprocessor communications make it rather difficult to use effectively, and thus the programs are typically rather complex, making them a good test-bed for para-functional programming. Figure 4-8 shows a ring of size **n=4**.

The first example computes the product of a matrix **A** and vector **b**. We assume that (1) the vector length is **n**, the same as the number of processors in the ring, (2) initially the rows of **A** and the corresponding elements of **b** are distributed uniformly around the ring, and (3) we wish the result vector to be distributed in the same way. Although other data configurations are possible, this represents a "typical" situation in scientific computing.

Given this initial configuration, the basic idea behind our algorithm is to compute each inner-product one scalar multiplication at a time, while trying to minimize interprocessor communication. Since the **i**th inner product needs to end up on processor **i**, it makes sense to compute it there. To do this the **i**th processor needs the **i**th row of **A** and the entire vector **b**. Although it has all of the former, initially it has only the **i**th element of the latter. Thus initially the **i**th processor is able to multiply $A_{i,i}$ by b_i without having to perform any remote accesses. The vector **b** can then be "shifted" one position around the ring so that the **i**th processor has the **(i+1 mod n)**th element of **b** -- this requires exactly **n** interprocessor "messages." It can then multiply that element by $A_{i,i+1 \bmod n}$, and add the result to the "running sum" for the **i**th inner product. This process of shifting **b** and computing another piece of each inner product is repeated **n** times, after which the final set of inner products is returned in a vector.

---

[5] For an alternative (and simpler) way to accomplish this annotation, see comment 5 in Section 6.

```
{ $result prod 0 (mkv n zerov);
  zerov i == 0 $on i;

  prod k acc == k=n -> acc,
                { $result prod (k+1) #(mkv n sum);
                  sum i == (A i ((i+k)\n) * (blist i ((i+k)\n))
                           + acc i) $on i };

  blist  == mkv n bvec;
  bvec i == { $result mkv n bproc;
              bproc j == j=i -> b i,
                         blist ((i+1)\n) j } $on i;
}
```

Figure 4-7: Matrix-vector product annotated for a ring topology

The first step in writing a ParAlfl program for this is to take care of "shifting" the vector **b**.[6] This can be done in several ways, but the way we have chosen is to create a copy of **b** on each processor, whose ith element is taken directly from **b**, and whose other elements are received "by demand" from each processor's immediate (say, left) neighbor (whose pid is **(i+1) mod n**). This causes the desired "shifting" effect, and is captured by the vector **blist** in Figure 4-7 -- each of its elements is a copy of **b** having the desired properties. Note the recursive nature of the definition.

The only remaining step is to design a function to index through **A** and **blist**, performing multiplications, and keeping a running sum for each inner product. The function **prod** in Figure 4-7 accomplishes this -- **k** is the "index" and **acc** is the "accumulator" containing the running sum (represented as a vector spread over the **n** processors). Initially the running sum **acc** is a vector of all zeros (**zerov**), and at the end (i.e., after **n** iterations of **prod**) it is the desired matrix-vector product. Note that the second argument to **prod** is "eagerly" evaluated.

It is interesting to note that with this solution it is possible for one processor to "get ahead" of another, but since each processor ends up with its own copy of **b**, this is not a problem. The structure of **blist** is such that only $\mathbf{n}^2$ immediate-neighbor transactions are incurred, no matter what the "synchronization" properties between the processors are.

However, it could be argued that there is overhead in permitting such overlap of computation, such as the maintenance of "process descriptors" or other implementation mechanisms. In such cases the user may wish to explicitly "synchronize" each "step" of the computation, which has the added advantage of being intuitively easier to understand, and can be done quite simply by modifying the definition of **prod** to:

---

[6] Remember that in ParAlfl vectors and matrices look syntactically just like functions; `A i j`, for example, returns the `[i,j]`th element from the matrix A.

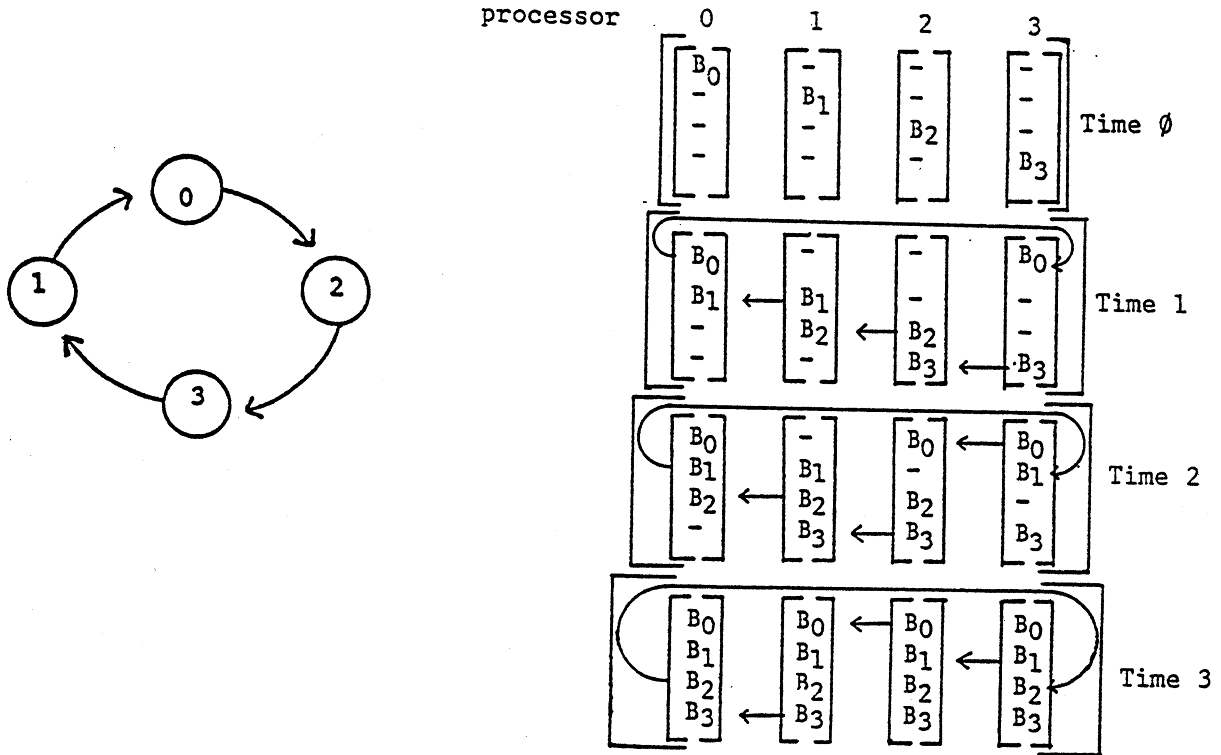


Figure 4-8: The construction of blist (n=4)

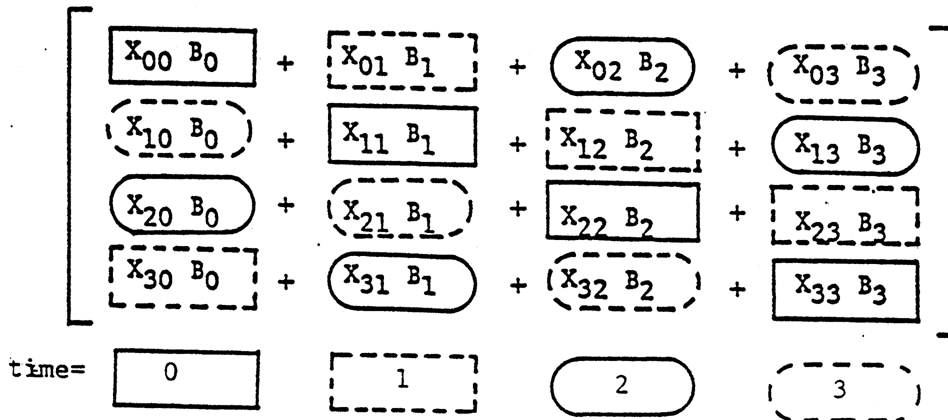


Figure 4-9: The computation of the result vector (n=4)

```
prod k acc == k=n -> acc,
              { $result strictify new-acc (prod (k+1) new-acc);
                new-acc == mkv n sum;
                sum i == ((A i ((i+k)\n)) * (blist i ((i+k)\n))
                          + acc i) $on i }
```

where **strictify** is as defined in Section 3.3. This version of **prod** simply ensures that **new-acc** is completely evaluated before the recursive call is made.

With such synchronization it is now possible to draw two figures that help describe the overall behavior of the program. First, Figure 4-8 shows the "filling in" of **blist**'s values during each "step" of the computation. Second, Figure 4-9 shows in which time step each component of the final matrix-vector product is computed.

4.4. Solution to Upper Triangular Block Matrix

The next example solves for the vector **x** in the equation **Ux=b**, where **U** is an upper triangular block matrix. As for the previous example, we use a ring of size **n**, which is also the vector length, and each row of **U** and the corresponding elements of **b** are distributed uniformly across the ring. We wish the solution vector **x** to be distributed as well.

Solve $U_{n-1,n-1}x_{n-1}=b_{n-1}$ on processor n-1. (step 1)
For i=n-2 down to 0 do:
Begin For j=i down to 0 do in parallel on processor j:
  $b_j := b_j - U_{j,i+1} * x_{i+1}$. (step 2)
  Solve $U_{i,i}x_i=b_i$ on processor i. (step 3)
End.

Figure 4-10: Algorithm for solving **Ux=b** on a ring

Figure 4-10 outlines the conventional "block row" algorithm for solving **Ux=b**. Initially, the last element $\mathbf{x}_{n-1}$ of the solution vector is computed on processor **n-1** (step 1). Then, a "back-substitution" step takes place in parallel on the remaining **n-1** processors (step 2). Completing this, the next-to-the-last element of the solution ($\mathbf{x}_{n-2}$) can be computed (step 3). This alternating process (steps 2 and 3) of solving and back-substituting continues until all elements of **x** are found.

```
{ $result iter (n-2) [lastx] b $on n-2;

  lastx == uxb (u (n-1) (n-1)) (b (n-1)) $on n-1;
  uxb u b == b/u

  iter 0 x b == x;
  iter i x b == { $result iter (i-1) (sol^x) newb $on i-1;
                  newb == mkv i backsub;
                  backsub j == (b j) - (U j (i+1))*(hd x) $on j;
                  sol == uxb (U i i) (newb i)   };
}
```

Figure 4-11: Program for solving **Ux=b**, annotated for a ring

A ParAlfl program for this algorithm is shown in Figure 4-11. For simplicity we assume that the blocks are of size 1 (and thus are represented simply as scalar quantities), but conceptually any size block could be used. For block size 1, the solution to $U_{i,i}x_i=b_i$ is simply $x_i= b_i/U_{i,i}$, and is captured by the function **uxb**.

The solution vector is represented as a list, since it is computed element-by-element with intervening back-substitution steps. The last element, and first to be computed, is **lastx** (step 1 of the previous description). The function **iter** accomplishes the iteration of steps 2 and 3, and collects the elements of the solution in the parameter **x**. Note how **iter** recursively performs the back-substitution by creating a new version of **b** (**newb**) on each iteration. This vector decreases in length by 1 on each iteration, and is computed in parallel as the previous description of the algorithm suggests (indeed, all of the parallelism in the program is manifested here). It is then used to compute **sol**, the next element of the solution vector. Figure 4-12 shows the temporal progression of the program, alternating between computing a new element of the solution and doing a back-substitution.

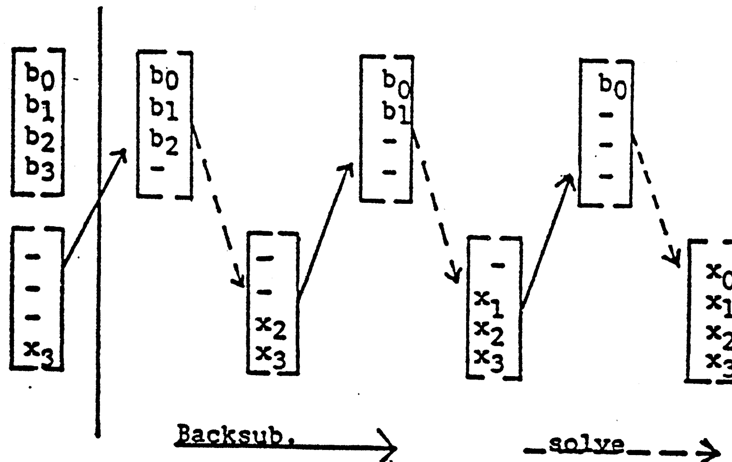


Figure 4-12: Time sequence of program to solve Ux=b

Note in this example and the previous one (the matrix-vector product) that **$self** is not used at all; thus both programs are essentially *statically* mapped to the ring. In this example the vector **b** and subsequent vectors **newb** are uniformly distributed around the ring. There is no need for any processor to have more than one element from any of these vectors, since each processor is "dedicated" to a particular element. No special synchronization constructs are needed since the "call-by-need" semantics achieves the necessary synchronization automatically.

Note finally that if true *block* matrices and vectors are used, the only change necessary is the redefinition of the functions **uxb** and **backsub**.

5. A Formal Semantics for ParAlfl

Hopefully the reader at this point has a good intuitive feel for the operational semantics of ParAlfl. However, we have glossed over many non-trivial details. For example: (1) In a mapped expression, where is the pid expression evaluated? (2) Where is an unmapped expression evaluated that appears at the top level? (3) The identifier **$self** in a function body is evaluated when the function is *applied*, not when it is created, but what exactly does that mean, especially in the context of curried functions? One approach to providing these details is to anticipate as many of the above questions as possible, and answer each of them in turn. This informal approach is unfortunately error-prone and susceptible to ambiguity because of the English language. A better approach is to give a formal denotational semantics that captures the desired operational properties, and that is what we do here.

For the following we assume that the reader is familar with the standard notational conventions for denotational semantics, such as outlined in [10]. For clarity we also assume that the following syntactic transformations have occurred on the ALFL or ParAlfl source program:

- The equations defining curried functions, including pattern-matching, have been mapped into lambda expressions. For example:

  fac 0 == 1;
  ' n == n*fac(n-1);

  will be converted into:

  fac == λn. n=0 -> 1, n*fac(n-1)

- Lists have been converted into an equivalent infix cons form. For example, **[x,y,z]** is converted into **x^y^z^[]**.

5.1. Standard Interpretation

To give a formal operational semantics for ParAlfl we first need to define its standard functional semantics, which after the above syntactic transformations is relatively straightforward. It is captured by the semantic function **E: Exp->Env->D**, where **Exp** is the syntactic domain of ParAlfl expressions, **Env** is the domain of environments that map identifiers to values, and **D** is the standard domain of expressible values. For convenience we will omit the details of the syntactic and semantic domains, as well as errors; this will allow us to concentrate on the issues of mapping. We adopt the normal convention of enclosing syntactic objects in double-brackets, and thus **E** ⟦**exp**⟧ **e** denotes the value of **exp** given environment **e**, and is defined by:

$$\mathbf{E}\llbracket\mathbf{constant}\rrbracket\,\mathbf{e} = \mathbf{B}\llbracket\mathbf{constant}\rrbracket$$

where B maps constants to their semantic counterpart.

$$\mathbf{E}\llbracket\mathbf{id}\rrbracket\,\mathbf{e} = \mathbf{e}\llbracket\mathbf{id}\rrbracket$$

$$\mathbf{E}\llbracket\mathbf{unop\ e1}\rrbracket\,\mathbf{e} = \mathbf{B}\llbracket\mathbf{unop}\rrbracket\ (\mathbf{E}\llbracket\mathbf{e1}\rrbracket\,\mathbf{e})$$

$$\mathbf{E}\llbracket\mathbf{e1\ binop\ e2}\rrbracket\,\mathbf{e} = \mathbf{B}\llbracket\mathbf{binop}\rrbracket\ (\mathbf{E}\llbracket\mathbf{e1}\rrbracket\,\mathbf{e})\ (\mathbf{E}\llbracket\mathbf{e2}\rrbracket\,\mathbf{e})$$

$$\mathbf{E}\llbracket\mathbf{pred\text{->}con,alt}\rrbracket\,\mathbf{e} = \mathbf{if}\ (\mathbf{E}\llbracket\mathbf{pred}\rrbracket\,\mathbf{e})\ \mathbf{then}\ (\mathbf{E}\llbracket\mathbf{con}\rrbracket\,\mathbf{e})\ \mathbf{else}\ (\mathbf{E}\llbracket\mathbf{alt}\rrbracket\,\mathbf{e})$$

$$\mathbf{E}\llbracket\lambda\mathbf{x.\ exp}\rrbracket\,\mathbf{e} = \lambda\hat{\mathbf{x}}.\ \mathbf{E}\llbracket\mathbf{exp}\rrbracket\ \mathrm{e}[\hat{\mathrm{x}}/\mathrm{x}]$$

$$\mathbf{E}\llbracket\mathbf{e1\ e2}\rrbracket\,\mathbf{e} = (\mathbf{E}\llbracket\mathbf{e1}\rrbracket\,\mathbf{e})\ (\mathbf{E}\llbracket\mathbf{e2}\rrbracket\,\mathbf{e})$$

$$\begin{aligned}\mathbf{E}\llbracket\{\ \mathbf{f1} &= \mathbf{e1}\\ &\cdots\\ \mathbf{fn} &= \mathbf{en}\\ \mathbf{\$result\ exp}\ \}\rrbracket\,\mathbf{e} &= \mathbf{E}\llbracket\mathbf{exp}\rrbracket\,\mathbf{e'}\\ &\quad \mathbf{whererec\ e'} = \mathbf{e}[\mathbf{E}\llbracket\mathbf{e1}\rrbracket\,\mathbf{e'}\ /\ \mathbf{f1},\\ &\qquad\qquad\cdots\\ &\qquad\qquad \mathbf{E}\llbracket\mathbf{en}\rrbracket\,\mathbf{e'}\ /\ \mathbf{fn}]\end{aligned}$$

$$\begin{aligned}\mathbf{E}\llbracket\mathbf{exp\ \$on\ pid}\rrbracket\,\mathbf{e} = {}&\mathbf{if}\ (\mathbf{E}\llbracket\mathbf{pid}\rrbracket\,\mathbf{e}) = \bot\ \mathbf{then}\ \bot\\ &\mathbf{else}\ (\mathbf{E}\llbracket\mathbf{exp}\rrbracket\ \mathbf{e}[\mathbf{E}\llbracket\mathbf{pid}\rrbracket\,\mathbf{e}/\mathbf{\$self}])\end{aligned}$$

$$\mathbf{E}\llbracket\#\mathbf{exp}\rrbracket\,\mathbf{e} = \mathbf{E}\llbracket\mathbf{exp}\rrbracket\,\mathbf{e}$$

The semantics of a complete ParAlfl program (which must be either an equation-group or a mapped expression whose body is an equation group) is given by the semantic function **Ep**, defined by:

$$\mathbf{Ep}\llbracket\mathbf{program}\rrbracket = \mathbf{E}\llbracket\mathbf{program}\rrbracket\,\mathbf{initial\text{-}env}$$

where **initial-env** contains bindings for primitive functions such as **plus**, **and**, etc., a default value for **$self**, and constants such as **true** and **false**.

5.2. Determinacy Theorem Revisited

We return now to Theorem 1, which we restate formally below:

> **Theorem 1:** (Formal) Let **P** be a ParAlfl program in which (1) the identifier **$self** appears only in pid expressions and (2) for each pid expression **pid** in environment **e**, $\mathbf{E}\llbracket\mathbf{pid}\rrbracket\,\mathbf{e} \neq \bot$. Further, let **P'** be the same ParAlfl program but with all occurrrences of mapped expressions of form **exp $on pid** replaced with the body **exp**, and all occurrences of eager expressions of form **#exp** replaced with **exp**. Then $\mathbf{Ep}\llbracket\mathbf{P}\rrbracket = \mathbf{Ep}\llbracket\mathbf{P'}\rrbracket$.

Proof. We must prove that $\mathbf{E}\llbracket\mathbf{P}\rrbracket\,\mathbf{initial\text{-}env} = \mathbf{E}\llbracket\mathbf{P'}\rrbracket\,\mathbf{initial\text{-}env}$. We do this by structural induction on **P**. Consider any subexpression **sexp** and its "stripped" version **sexp'**. For all but mapped expressions and eager expressions, it is clear that $\mathbf{E}\llbracket\mathbf{sexp}\rrbracket\,\mathbf{e} = \mathbf{E}\llbracket\mathbf{sexp'}\rrbracket\,\mathbf{e}$, since **$self** will never be evaluated. We consider mapped expressions and eager expressions below.

Suppose **sexp** = **exp $on pid** and thus **sexp'** = **exp**. Then:

$$\mathbf{E}[\![\mathbf{sexp}]\!]\,\mathbf{e} = \mathbf{if}\ (\mathbf{E}[\![\mathbf{pid}]\!]\,\mathbf{e}) = \bot\ \mathbf{then}\ \bot$$
$$\mathbf{else}\ (\mathbf{E}[\![\mathbf{exp}]\!]\ \mathbf{e}[\mathbf{E}[\![\mathbf{pid}]\!]\,\mathbf{e}/\$\mathbf{self}])$$

But we are given that $\mathbf{E}[\![\mathbf{pid}]\!]\,\mathbf{e} \neq \bot$, so: $\mathbf{E}[\![\mathbf{sexp}]\!]\,\mathbf{e} = \mathbf{E}[\![\mathbf{exp}]\!]\ \mathbf{e}[\mathbf{E}[\![\mathbf{pid}]\!]\,\mathbf{e}/\$\mathbf{self}]$. Furthermore, since **$self** will never be evaluated, we have that $\mathbf{E}[\![\mathbf{sexp}]\!]\,\mathbf{e} = \mathbf{E}[\![\mathbf{exp}]\!]\,\mathbf{e} = \mathbf{E}[\![\mathbf{sexp'}]\!]\,\mathbf{e}$.

The remaining case is an eager expression. Suppose **sexp** = **#exp** and thus **sexp'** = **exp**. Directly from the semantic equations for **E**, we then have that $\mathbf{E}[\![\mathbf{sexp}]\!]\,\mathbf{e} = \mathbf{E}[\![\mathbf{exp}]\!]\,\mathbf{e} = \mathbf{E}[\![\mathbf{sexp'}]\!]\,\mathbf{e}$. This covers all cases and thus the theorem holds. □

5.3. Execution Trees

The operational semantics that we wish to capture is a notion of "where" (i.e., on which processor) an expression will be evaluated. For each expression **exp** we associate an *execution tree* that reflects the evaluation history of **exp**. Intuitively, the root of **exp**'s execution tree **t** is labelled with the processor on which **exp** will be executed, and each immediate subtree of **t** is the execution tree of each immediate subexpression of **exp**. The notation $\mathbf{pid{:}}\langle t_1, t_2, \ldots, t_n\rangle$ denotes an execution tree whose root is labelled **pid** and whose children are the sub-trees t_1 through t_n. $\mathbf{pid{:}}\langle\rangle$ is a leaf node with label **pid**.

An execution tree may also be a lambda expression, corresponding to an expression that evaluates to a function under the standard interpretation. This is called an *abstracted execution tree*, and it may be applied to another tree, yielding a third. The application on processor **cp** of execution tree **t1** to an execution tree **t2** whose value is **v2**, is written "**t1 t2 v2 cp**," and is defined by:

- if **t1** = $\mathbf{p{:}}\langle\mathbf{t1},\ldots,\mathbf{tn}\rangle$, $n \geq 1$, then **tn t2 v2 cp**.
- if **t1** = λ**t v p. body** then a standard beta-reduction occurs.
- else error (i.e., **t1** does not represent a function).

In other words, the application "trickles down" the right subtrees until it encounters a lambda expression at the right-most leaf, in which case a normal function application occurs.

5.4. Semantics of Mapped Expressions

We now define a semantic function **T** such that $\mathbf{T}[\![\mathbf{exp}]\!]$ **cp te e** denotes the execution tree for the ParAlfl expression **exp** given the standard environment **e**, the "tree environment" **te**, and the currently executing processor **cp**. A tree environment maps identifiers to execution trees. The semantic function **T** is defined by:

```
T⟦constant⟧ cp te e = cp:<>

T⟦id⟧ cp te e = cp:<te⟦id⟧>

T⟦unop e1⟧ cp te e = cp:<T⟦e1⟧ cp te e>

T⟦e1 binop e2⟧ cp te e = cp:<T⟦e1⟧ cp te e, T⟦e2⟧ cp te e>

T⟦pred->con,alt⟧ cp te e = cp:<T⟦pred⟧ cp te e,
                               if E⟦pred⟧e then T⟦con⟧ cp te e
                               else T⟦alt⟧ cp te e>

T⟦λx. exp⟧ cp te e = cp:<λ tx vx cp'.
                          T⟦exp⟧ cp' te[tx/x] e[vx/x]>

T⟦e1 e2⟧ cp te e = cp:<T⟦e1⟧ cp te e,
                       T⟦e2⟧ cp te e,
                       (T⟦e1⟧ cp te e) (T⟦e2⟧ cp te e) (E⟦e2⟧e) cp>

T⟦{ f1 = e1
    ...
    fn = en
    $result exp }⟧ cp te e = T⟦exp⟧ cp te' e'
                              whererec te' = te[T⟦e1⟧ cp te' e' / f1,
                                                ...
                                                T⟦en⟧ cp te' e' / fn]
                                   and e' = e[E⟦e1⟧e' / f1,
                                              ...
                                              E⟦en⟧e' / fn]

T⟦exp $on pid⟧ cp te e = cp:<T⟦pid⟧ cp te e[cp/$self],
                             T⟦exp⟧ (E⟦pid⟧ e[cp/$self]) te e[cp/$self]>

T⟦#exp⟧ cp te e = T⟦exp⟧ cp te e
```

The first equation simply states that the value of a constant is computed where it is found. The second equation reflects the fact that the value of an identifier is "moved" to the current processor **cp**, but that it is evaluated on the processor on which it was defined, which is information contained in the tree environment **te**. The next three equations define execution trees for primitive functions in ParAlfl, including the conditional. The next two treat lambda abstraction and function application in turn; note how an abstracted execution tree is built, and how it is expanded when it is applied. Also note that in an application there are separate subtrees for the function, its argument, and the result of the application. The next equation gives the execution tree for an equation group; it is here that the two environments **te** and **e** are updated with new execution trees and functional values, respectively. The last two equations define the behavior of mapped and eager expressions. For a mapped expression **exp $on pid**, clearly **exp** will be evaluated on processor **pid**. However, that requires first computing the *value* of **pid**, which is done on processor **cp**. Note that the value is expressed as a call to **E** in an environment in which **$self** has the value **cp**.

The only remaining question is what the execution tree of a program might be. A program can be either an equation group or a mapped expression whose body is an equation group. We thus define the semantic function **Tp** as:

Tp ⟦**program**⟧ = **T** ⟦**program**⟧ **default-pid initial-tenv initial-env**

where **default-pid** is the name of some root processor on which an unmapped program is defined to execute, **initial-tenv** contains the execution trees for pre-defined values, and **initial-env** is as described earlier. Note that **initial-env** ⟦**$self**⟧ = **default-pid.**

6. Conclusions and Future Work

We have presented a way to explictly map a functional program for execution on an arbitrary multiprocessor, as well as a method to control the "eager" evaluation of subexpressions. This expressive power is gained through annotations that, given some minor constraints, cannot change the functional behavior of the program. We feel that the resulting parallel programming methodology, that we refer to as "para-functional programming," is a simple yet powerful way to write parallel programs.

Once this methodology is understood, many variations of it become apparent, and alternative language design features present themselves. For example:

1. The dynamic variable **$self** is one of many that could be provided to the programmer. Consider, for example, the use of:

   - The dynamic variable **$load** to give the programmer a measure of the "processing load" on the currently executing processor (perhaps it would simply be a count of the number of tasks waiting in the process queue). Using this variable the programmer could implement a dynamic load-balancing strategy tailored for a particular application.

   - The dynamic variable **$memory-utilization** to provide information about how much memory is left on a node.

2. In addition to allowing one to map processes to processors, one could allow mapping data objects to storage devices or I/O channels. For example, the expression **lst $on tape-drive-2** might cause the list **lst** to be written out to a particular tape drive. The right combination of such annotations, in addition to an appropriate set of non-deterministic operators, could provide a powerful set of tools to build a "purely functional" operating system.

3. It may also be desirable to perform an operation on a data structure *at the processor on which the data structure resides*. For example, the special function **$location** could be used to return the pid of its argument's "home." Thus the expression **(f obj) $on ($location obj)** could be used to apply the function **f** to **obj** on whichever processor **obj** happens to reside.

4. One of the more standard approaches to obtaining parallelism from a functional program is to allow the compiler and run-time system to automatically decompose and dynamically distribute the program for execution on a suitable multiprocessor [16, 17, 23]. An obvious compromise is to combine this automatic approach with the more explicit para-functional approach. For example, the annotated expression **exp $on $arb** might indicate that **exp** is to be decomposed and distributed as the system sees fit, but that subexpressions internal to it may still be mapped using the para-functional approach when needed.

5. Our language design choices were somewhat arbitrary, and thus other choices may be more suitable for particular applications. One of the more obvious examples of this is in the use of # -- currently this annotation "parallelizes" the subexpression relative to, more or less, the most immediately surrounding syntactic form. Another possibility would to parallelize the subexpression relative to the innermost *equation* or *result clause*. Suppose that the annotation ! is used for this purpose, and consider the function **sift** defined in Section 4.2:

```
sift (p^rest) == { new == filter rest;
                   $result par new (p ^ (sift new $on right $self));
                   filter (n^l) == n\p=0 -> filter l, n ^ filter l }
```

With ! this could be rewritten as:

```
sift (p^rest) == { $result p ^ (sift !(filter rest) $on right $self);
                   filter (n^l) == n\p=0 -> filter l, n ^ filter l }
```

At least for this example, the use of ! is much simpler and clearer.

A final area left open for investigation is ways to effectively *implement* a para-functional programming language. Although we have made some progress in this area, the work is too premature to present here. Our goal is to implement a working system on an Intel iPSC 128-node hypercube multiprocessor. A future report will detail our success in this endeavor.

7. Acknowlegements

This research has benefited from useful discussions with Bob Keller at the University of Utah, and Joe Fasel and Elizabeth Williams at Los Alamos National Laboratory.