Para-Functional Programming

Paul Hudak
Yale University

This methodology treats a multiprocessor as a single autonomous computer onto which a program is mapped, rather than as a group of independent processors.

The importance of parallel computing hardly needs emphasis. Many physical problems and abstract models are seriously compute-bound, since sequential computer technology now faces seemingly insurmountable physical limitations. It is widely believed that the only feasible path toward higher performance is to consider radically different computer organizations, in particular ones exploiting parallelism. This argument is indeed rather old now, and considerable progress has been made in the construction of highly parallel computers.

One of the simplest and most promising types of parallel machines is the well-known *multiprocessor architecture*, a collection of autonomous processors with either shared or distributed memory that are interconnected by a homogeneous communications network and usually communicate by sending messages. The interest in machines of this type is not surprising, since not only do they avoid the classic "von Neumann bottleneck" by being effectively decentralized, but they are also extensible and in general quite easy to build. Indeed, more than a dozen commercial multiprocessors either are now or will soon be available.

Although designing and building multiprocessors has proceeded at a dramatic pace, the development of effective ways to program them has generally not. This is an unfortunate state of affairs, since experience with sequential machines tells us that software development, not hardware development, is the most critical element in a system's design. The immense complexity of parallel computation can only increase our dependence on software. Clearly we need effective ways to program the new generation of parallel machines.

In this article I introduce *para-functional programming*, a methodology for programming multiprocessor computing systems. It is based on a functional programming model augmented with features that allow programs to be mapped to specific multiprocessor topologies. The most significant aspect of the methodology is that it treats the multiprocessor as a single autonomous computer onto which a program is mapped, rather than as a group of independent processors that carry out complex communication and require complex synchronization. In more conventional approaches to parallel programming, the latter method of treatment is often manifested as processes that cooperate by message-passing. However, such notions are absent in para-functional programming; indeed, a single language and evaluation model can be used from

0018-9162/86/0800-0060$01.00

problem inception, to prototypes targeted for uniprocessors, and ultimately to realizations on a parallel machine.

Functional programming and parallel computing

The future of parallel computing depends on the creation of simple but effective parallel-programming models (reflected in appropriate language designs) that make the details of the underlying architecture transparent to the user. Many researchers feel that conventional imperative languages are inadequate for such models, since these languages are intrinsically tied to the "word-at-a-time" von Neumann machine model.[1] Extending such a sequential model to the parallel world is like putting on a shoe that doesn't fit. It makes more sense to use a language with a nonsequential semantic base.

One of the better candidates for parallel computing is the class of *functional languages* (also known as *applicative* or *dataflow* languages). In a functional language, no side effects (such as those caused by an assignment statement) are permitted. The lack of side effects accounts at least partially for the well-known *Church-Rosser Property,* which essentially states that no matter what order of computation is chosen in executing a program, the program is guaranteed to give the same result (assuming termination). This marvelous determinacy property is invaluable in parallel systems. It means that programs can be written and debugged in a functional language on a sequential machine, and then the *same* programs can be executed on a parallel machine for improved performance. The key point is that in functional languages the parallelism is implicit and supported by their underlying semantics. There is generally no need for special message-passing constructs or other communications primitives, no need for synchronization primitives, and no need for special "parallel" constructs such as "parbegin...parend."

On the other hand, doing without assignment statements seems rather radical. Yet clearly the assignment statement is an artifact of the von Neumann computer model and is not essential to the most abstract form of computation. In fact, a major goal of high-level language design has been the introduction of *expressions,* which transfer the burden of generating sequential code involving assignments from programmer to compiler. Functional languages simply carry this goal to the extreme: *Everything* is an expression. The advantages of the resulting programming style have been well-argued elsewhere,[1-2] and will not be repeated here. However, I wish to emphasize the following point: Although most experienced programmers recognize the importance of minimizing side effects, the importance of doing so in a parallel system is intensified significantly, due to the careful synchronization required to ensure correct behavior when side effects are present. Without side effects, there is no way for concurrent portions of a program to affect one another adversely—this is simply another way of stating the Church-Rosser Property.

The use of functional languages for parallel programming is really nothing new. Such use has its roots in early work on dataflow and reduction machines, in the course of which many functional languages were developed simultaneously with the design of new parallel architectures. Consider, for example, J. B. Dennis's dataflow machine and the language VAL, Arvind's U-interpreter and the language ID, A. L. Davis's dataflow machine DDM1 and the language DDN, and R. M. Keller's reduction machine AMPS and the language FGL. Such work on automatically decomposing a functional program for parallel execution continues today, and includes Rediflow[3] and my own work on serial combinators.[4]

The aforementioned systems automatically extract parallelism from a program and dynamically allocate the resultant tasks for parallel execution. But what about a somewhat different scenario—one in which the programmer knows the optimal mapping of his or her program onto a particular multiprocessor? One cannot expect an automated system to determine this optimal mapping for all program-processor combinations, so it is desirable to provide the user with the ability to express the mapping *explicitly.* (The need for this ability often arises, for example, in scientific computing, where many classic algorithms have been redesigned for optimal performance on particular machines.) As it stands, almost no languages provide this capability.

ParAlfl provides a mechanism for mapping a program onto an arbitrary multiprocessor.

ParAlfl is a functional language that provides a simple yet powerful mechanism for mapping a program onto an arbitrary multiprocessor. The mapping is accomplished by *annotating subexpressions* so as to show the processor on which they will be executed. With annotations, the mapping can be done in such a way that the program's functional behavior is not altered; that is, the program itself remains unchanged. The resulting methodology is referred to as *para-functional programming,* since it provides not only a much-needed tool for expressing parallel computation, but also an operational semantics that is truly "extra," or "beyond" the functional semantics of the program. It is quite powerful, for several reasons:

- It is very flexible. Not only is para-functional programming easily adapted to any functional language, but also any network topology can be captured by the notation, since no a priori assumptions are made about the structure of the physical system. All the benefits of conventional scoping disciplines are available to create modular programs that conform to the topology of a given machine.
- 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.
- With some minor 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 that ignores the annotations, and

An important semantic feature of ParAlfl is lazy evaluation.

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.

ParAlfl: a simple para-functional programming language

ParAlfl forms the testbed of Yale's para-functional programming research. It was derived from a functional language called ALFL,[5] which is similar in style to several modern functional languages, including SASL (and its successors KRC and Miranda),[6] FEL,[7] and Lazy ML.[8]

The base language. To make ParAlfl accessible to a broader audience, the base language, as shown here, was changed somewhat; for example, the arguments in function calls are "tupled" rather than "curried." The interested reader is referred to Henderson[2] and to Darlington, Henderson, and Turner[9] for a more thorough treatment of the functional programming paradigm. The salient features of the base language are

- Block-structuring is used, and takes the form of an *equation group* with the following configuration:

```
{ f1(x1,...,xk1) = = exp1;
  f2(x1,...,xk2) = = exp2;
  ...
  result exp;
  ...
  fn(x1,...,xkn) = = expn }
```

An equation group is simply a collection of mutually recursive equations (each defining a local identifier) together with a single *result clause* that expresses the value to which the equation group will evaluate (*result* is a reserved word). Equation groups are just expressions, and can thus be nested to an arbitrary depth.

- A double equal-sign ("= =") is used to distinguish equations from Boolean expressions of the form *exp1 = exp2*. The argument list is optional, allowing definitions of simple values, such as *x = = exp*. Since the equations are mutually recursive, and since ParAlfl is a lazy functional language, the order of the equations is irrelevant. All values are essentially evaluated "on demand."

- As in Lisp, the *list* is a fundamental data structure in ParAlfl. The operators ^ , ^^ , *hd*, and *tl* are like *cons*, *append*, *car*, and *cdr*, respectively, in Lisp. ^ and ^^ build lists: *a^l* is the list whose first element is *a* and the rest is just the list *l*, and *l1^^l2* is the list resulting from appending the lists *l1* and *l2* together. *Hd* and *tl* decompose lists: *hd(a^l)* returns *a*, and *tl(a^l)* returns *l*. A *proper list* (one ending in "nil") can be constructed with square brackets, as in *[a,b,c]*, which is equivalent to *a^b^c^[]* (^ is right associative). Lists are constructed lazily.

- ParAlfl has functional arrays. The equation *v = = mka(d,f)* (*mka* is short for "make array") defines a vector *v* of *d* values, indexed from 1 to *d*, such that the *i*th element *v[i]* is the same as *f(i)*. Generally, the equation *a = = mka(d1,d2, ...,dn,f)* defines an *n*-dimensional array *a* such that *a[i1,...,in] = f(i1,...,in)*. Arrays are constructed lazily, although the elements are computed in parallel. (See the section entitled "Eager Expressions," below.) In an earlier article on para-functional programming[10] arrays were defined as being non-lazy, or *strict*. In reality, both kinds of array construction are provided in ParAlfl.

An important semantic feature of ParAlfl is *lazy evaluation.** That is, expressions are evaluated on demand instead of according to some syntactic rule, such as the order of identifier bindings. For example, one can write

```
{ a = = b*b;
  b = = 2;
  result f(a,b);
  f(x,y) = = if p then y else x+y }
```

Note that *a* depends on *b*, yet is defined *before b*. Indeed, the order of these equations and the result clause is totally irrelevant. Note further that the function *f* does not use its first argument if *p* is true. Thus, in the call *f(a,b)*, the argument *a* is never evaluated (that is, the multiplication *b*b* never happens) if *p* is true.

An often highlighted feature of lazy evaluation is its ability to express unbounded data structures, or *infinite lists*. For example, an infinite list of the squares of the natural numbers can be defined by

```
{ result squares(0);
  squares(n) = = n*n ^ squares(n+1) }
```

However, an important but often overlooked advantage of lazy evaluation is simply that it frees the programmer from extraneous concerns about the order of evaluation of expressions. Being freed from such concerns is very liberating for programming in general, but is especially important in parallel programming because over-specifying the order of evaluation can limit the potential parallelism.

Mapped expressions. A program can be mapped onto a particular multiprocessor architecture through the use of *mapped expressions*. These form one of the two classes of extensions (annotations) to the base language. (The other class is made up of *eager expressions*, which are described below.) Mapped expressions have the simple form

```
exp $on proc
```

which declares that *exp* is to be computed on the processor identified by *proc* (*on proc* is prefixed with $ to emphasize that *$on proc* is an annotation). The expression *exp* is the *body* of the mapped expression, which is to say, it 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. Without loss of generality, we will assume in all examples below that processor IDs, or *pids*, are integers and that there is some predefined mapping from those integers to the physical processors they denote. For example, a tree of

*Lazy evaluation is closely related to the *call-by-name* semantics of Algol, but is different in that once an expression is computed, its value is retained. In function calls, lazy evaluation is sometimes referred to as *call-by-need evaluation*.

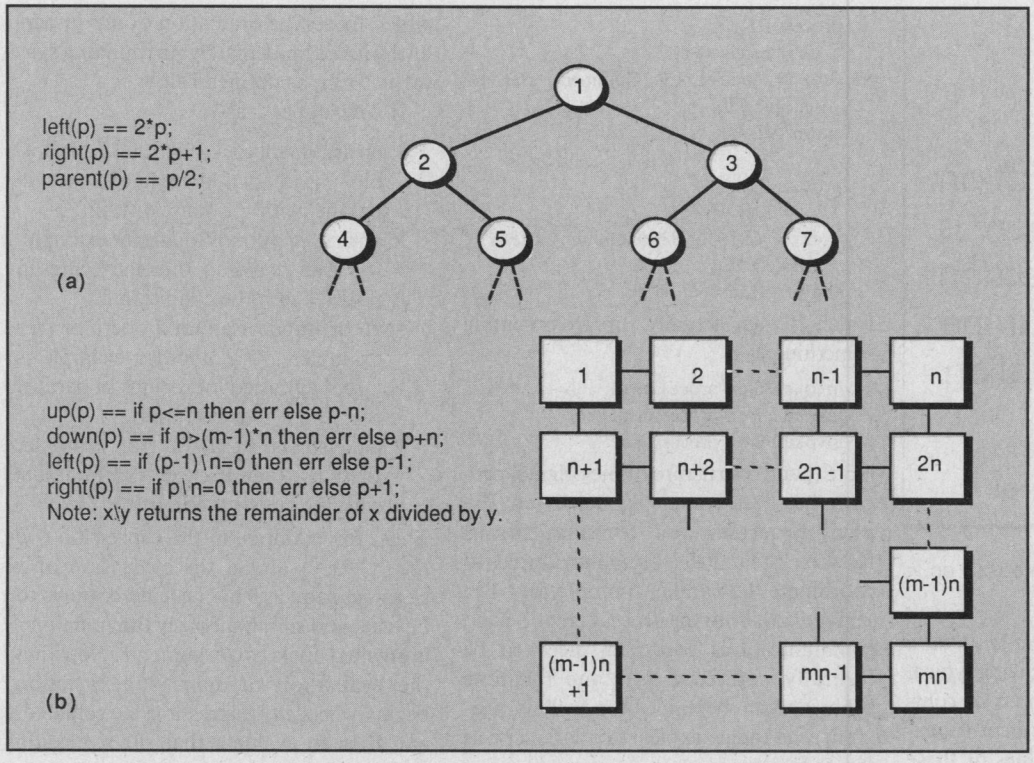

Figure 1. Two possible network topologies: infinite binary tree (a), and finite mesh of size $m \times n$ (b). Listed with each topology are functions that map pids to neighboring pids.

processors might be numbered as shown in Figure 1(a) and a mesh as shown in Figure 1(b). The advantage of using integers is that the user can manipulate them with conventional arithmetic primitives; for example, Figure 1 also defines functions that map pids to neighboring pids. However, a safer discipline might be to define a pid as a unique data-type, and to provide primitives that enable the user to manipulate values having that type.

Simple examples of mapped expressions. Consider the program fragment

```
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 can do so by annotating them, 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. It would be nice, for example, if we were able to refer to a processor with respect to the currently executing one. ParAlfl provides this ability through the reserved identifier *$self*, which when evaluated returns the pid of the currently executing processor. Using *$self* we can be more creative. For example, suppose we have a mesh or tree of processors as shown in Figure 1; we can then write

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

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

We can describe the behavior of *$self* more precisely as follows: *$self* is bound implicitly by mapped expressions; thus, in

```
exp $on pid
```

$self has the value *pid* in *exp*, unless it is further modified by a nested mapped expression. Although *$self* is a reserved word that cannot be redefined, this implicit binding can be best explained with the following analogy:

```
exp $on pid
```

is like

```
{ $self = = pid; result exp }
```

However, the most important aspect of *$self* is that it is *dynamically bound* in function calls. Thus, in

```
{ result (f(a) $on pid1) + (f(b) $on pid2);
  f(x) = = x*x } $on pid3
```

$a*a$ is computed on processor pid_1, $b*b$ on processor pid_2, and the sum on processor pid_3. As before, an analogy is useful in describing this behavior:

```
f(x,y,z,) = = exp;
... f(a,b,c) ...
```

is like

```
f(x,y,z,$self) = = exp;
... f(a,b,c,$self) ...
```

In other words, all functions implicitly take an extra *formal parameter, $self,* and all function calls use the current value of *$self* as the value for the new *actual parameter.*

Although very powerful, *$self* is not always needed. Particular cases illustrating this are those in which 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 $mka(n,f)$ will produce a vector of squares, one on each of n processors, such that the ith processor contains the ith element (namely i^2). Further, suppose we have two vectors v and w and we wish to create a third that is the sum of the other two, but distributed over the n processors. This can be done very simply by

```
mka(n,g);
 g(i) = = (v[i] + w[i]) $on i
```

If v and w were already distributed in the same way, this would express the pointwise

The value of an "eager expression" is that of the expression without the annotation.

parallel summation of two vectors on n processors.

A note on lexical scoping and data movement. Consider the following typical situation: A shared value v is to be computed for use in two independent subexpressions, *e1* and *e2*; the values of these subexpressions are then to be combined into a single result. In a conventional language one might express this as something like

```
begin v := code-for-v;
      e1 := code-for-e1;
            (Comment: uses v)
      e2 := code-for-e2;
            (Comment: uses v)
      result := combine(e1,e2);
end;
```

and in ParAlfl one might write

```
{ v = = code-for-v;
  e1 = = code-for-e1; (Comment: uses v)
  e2 = = code-for-e2; (Comment: uses v)
  result combine(e1,e2) }
```

Both of these programs are very clear and concise.

But now suppose that this same computation is to begin and end on processor p, and the subexpressions *e1* and *e2* are to be executed in parallel on processors q and r, respectively. In a conventional language augmented with explicit process-creation and message-passing constructs, one might write the following program:

```
process P0;
  v := code-for-v;
  send(v,P1);
  send(v,P2);
  e1 := receive(P1);
  e2 := receive(P2);
  result := combine(e1,e2)
end-process;

process P1;
  v := receive(P0);
  e1 := code-for-e1; (Comment: uses v)
  send(e1,P0);
end-process;

process P2;
  v := receive(P0);
  e2 := code-for-e2; (Comment: uses v)
  send(e2,P0);
end-process;
```

which is then actually run by executing something like

```
invoke P0 on processor p;
invoke P1 on processor q;
invoke P2 on processor r;
```

Note that the structure of the original program has been completely destroyed. Explicit processes and communications between them have been introduced to coordinate the parallel computation. The semantics of both the process-creation and communications constructs need to be carefully defined before the run-time behavior can be understood. This program is no longer as clear nor as concise as the original one.

On the other hand, a ParAlfl program for this same task is simply

```
{ v = = code-for-v;
  e1 = = code-for-e1 $on q;
            (Comment: uses v)
  e2 = = code-for-e2 $on r;
            (Comment: uses v)
  result combine(e1,e2) } $on p
```

Note that if the three annotations are removed, the program is identical to the ParAlfl program given earlier! No communications primitives or special synchronization constructs are needed to send the value of v to processors q and r; standard lexical scoping mechanisms accomplish the data movement naturally and concisely. The values of *e1* and *e2* are sent back to processor p in the same way.

Eager expressions. The second form of annotation, the *eager expression,* arises out of the occasional need for the programmer to override the lazy-evaluation strategy of ParAlfl, since normally ParAlfl does not evaluate an expression until absolutely necessary. (This second type of annotation is not needed in a functional language with non-lazy semantics, such as pure Lisp, but as mentioned earlier, we prefer the expressiveness afforded by lazy semantics.) An eager expression has the simple form

```
#exp
```

which forces the evaluation of *exp* in parallel with its immediately surrounding syntactic form, as defined below:

If *#exp* appears as

- an argument to a function (for example, $f(x,\#y,z)$), then it executes in parallel with the function call.
- an arm of a conditional (for example, *if p then #x else y*), then it executes in parallel with the conditional.
- an operand of an infix operator (for example, *x^#y;* another example is *x and #y*), then it executes in parallel with the whole operation.
- an element of a list (for example, $[x,\#y,z]$), then it executes in parallel with the construction of the list.

Thus, for example, in the expression *if p then f(#x,y) else 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 evaluate that subexpression "eagerly" accomplishes nothing. For example, note the following equivalences:

```
if #p then x else y ≡ if p then x else y
#x and y ≡ x and y
#x + #y ≡ x + y
```

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 within lists. Consider, for example, the expression $[x,\#y]$; normally lists are constructed lazily in ParAlfl, so the values of x and y are not evaluated until selected. But with the annotation shown, y would be evaluated as soon as the list was demanded. As with arrays, however, the expression does not wait for the value of y to return a fully computed value. 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 that of the expression without the annotation. As with mapped expressions, the annotation only adds an operational semantics, and thus the user can invoke a nonterminating subcomputation, yet have the overall pro-

gram terminate. Indeed, in the above example, even should y not terminate, if only the first element of the list is selected for later use, the overall program may still terminate properly. The "runaway process" that computes y is often called an *irrelevant task,* and there exist strategies for finding and deleting such tasks at run time. Such considerations are beyond the scope of this article, although it should be pointed out that given an automatic task-collection mechanism there are real situations in which one may wish to invoke a nonterminating computation (an example of this is given in Hudak and Smith[10]).

A note on determinacy. All ParAlfl programs possess the following determinacy property:

> A ParAlfl program in which (1) the identifier $\$self$ appears only in pid expressions, and (2) all pid expressions terminate without error, is functionally equivalent to the same program with all of the annotations removed. That is, both programs return the same value.

(A formal statement and proof of this property depends on a formal denotational semantics for ParAlfl, which is beyond the scope of this article, but such semantics can be found in Hudak and Smith.[10])

The reason for the first constraint is that if the mapping annotations are removed, all remaining occurrences of $\$self$ have the same value, namely the pid of the root processor. Thus, removing the annotations may change the value of the program. The purpose of the second constraint should be obvious: If the system diverges or errs 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.

Although neither determinacy constraint is severe, there are practical reasons for wanting to violate the first one (that is, for wanting to use the value of $\$self$ in other than a pid expression). The most typical situation where this arises is in a nonisotropic 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 $\$self$.

```
{ result pfac(1,k) $on root;

  pfac(lo,hi) = =if lo = hi then lo
                else if lo = (hi – 1) then lo*hi
                     else { result (pfac(lo,mid) $on left($self)) *
                                   (pfac(mid + 1,hi) $on right($self));
                            mid = =(lo + hi)/2 };

  left(pe) = =if 2*pe > n then pe else 2*pe;
  right(pe) = =if 2 *pe > n then pe else 2*pe + 1;
  root = =1;
}
```

Figure 2. Divide-and-conquer factorial on finite tree.

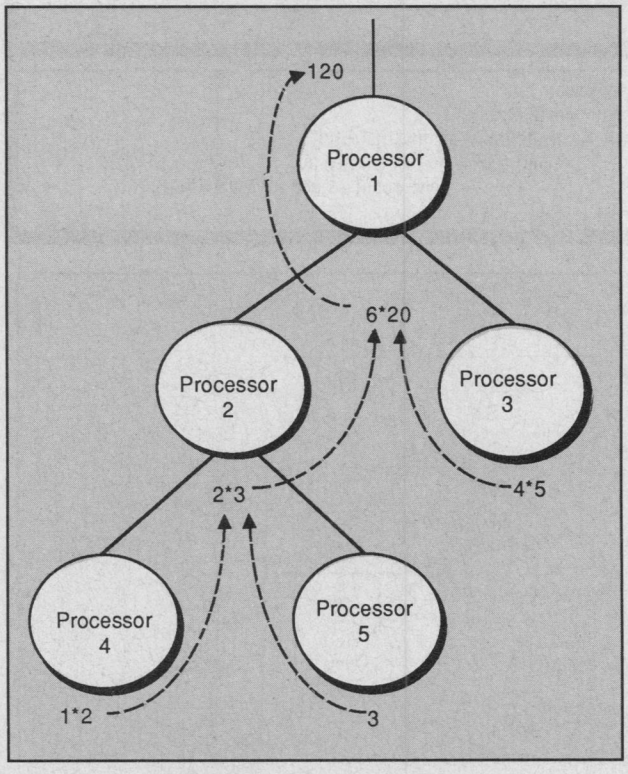

Figure 3. Dataflow for parallel factorial.

Sample application programs

In this section two simple examples are presented that highlight the key aspects of para-functional programming. Space limitations preclude the inclusion of examples that are more complex, but some can be found in Hudak and Smith[10] and Hudak.[11]

Parallel factorial. Figure 2 shows a simple parallel factorial program annotated for execution on a finite binary tree of $n = 2^d - 1$ processors. Although computing factorial, even in parallel, is a rather simple task, the example demonstrates several important ideas, and most other divide-and-conquer algorithms could easily fit into the same framework.

The algorithm is based on splitting the computation into two parts at each iteration and mapping the two subtasks onto the "children" of the current processor. 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 the section on "Mapped expressions," above). The functions *left* and *right* describe the network mapping necessary

```
{ result pfac(1,k) $on root;

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

  sfac(lo,hi,acc) = =if lo = hi then lo*acc
                     else 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. Parallel factorial, with unique behavior at leaves.

```
{ result xvect;
  xvect = =mka(n,x);
  x(i) = = { result (b[i] - sum(n,0)) / U[i,i];
             sum(j,acc) = =if j < i + 1 then acc
                          else sum(j - 1,acc + xvect[j]*U[i,j]) }
}
```

Figure 5. ParAlfl program to solve $Ux = b$.

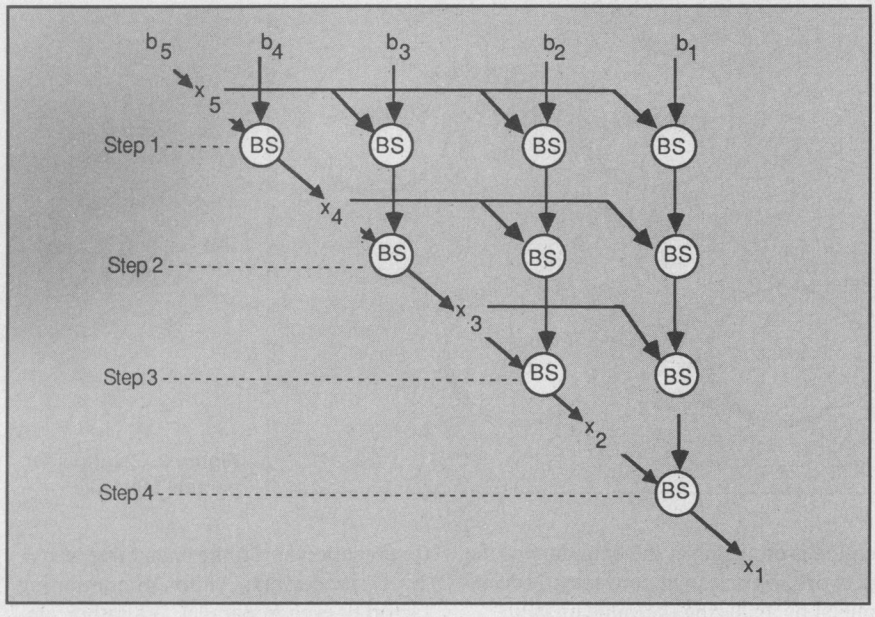

Figure 6. Dataflow for matrix problem. ("BS" stands for backsubstitution.)

for this topology, and Figure 3 shows the process-mapping and flow of data between processes when $k = 5$.

Note that the program in Figure 2 obeys the constraints required for determinacy, and thus the program returns the same value regardless of the annotations. Note further that with the mapping used, when processing reaches a leaf node all further calls to *pfac* are executed on the leaf processor. Routing functions of greater complexity could be devised that, for example, would reflect the computation upward once a leaf processor is reached. Alternatively, it might be desirable to use a more efficient factorial algorithm at the leaf nodes. An example of this is given in Figure 4, where the tail-recursive function *sfac* is invoked at the leaves. Determining that execution has reached a leaf processor requires inspection of *$self*, and thus the determinacy constraints are violated, yet the program still returns the same value regardless of the annotations. This constancy of values is, of course, often the case, but it cannot be guaranteed in general without the previously discussed constraints.

Solution to upper triangular block matrix. The next example is typical of problems encountered in scientific computing: The problem is to solve for the vector x in the matrix equation $Ux = b$, where U is an *upper triangular block matrix* (that is, a matrix whose elements are themselves matrices, and whose elements below the main diagonal contain all zeros). Algorithms using block matrices are especially suited to multiprocessors with nontrivial communications costs, since typically the subcomputations involving the submatrices can be done in parallel with little communication between the processors.

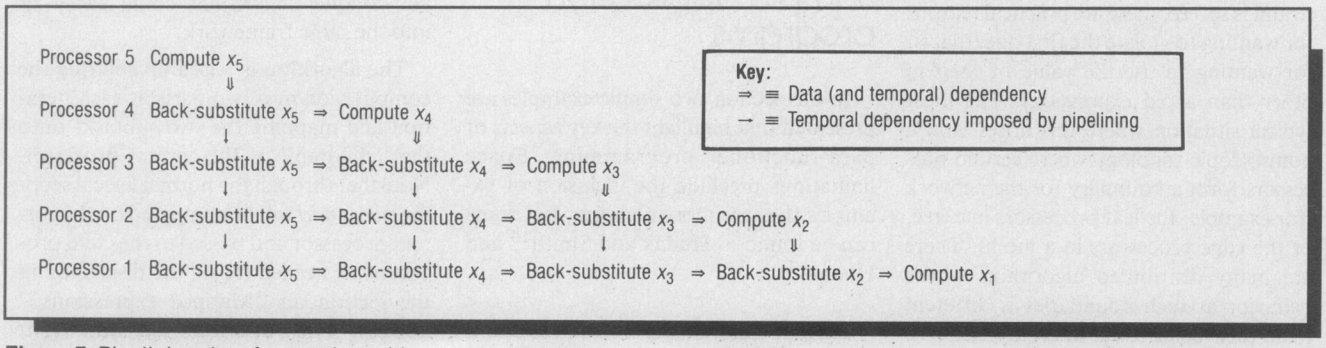

Figure 7. Pipelining data for matrix problem.

If we ignore parallelism at the moment and concentrate instead on a functional specification of this problem, it is easy to see from basic linear algebra that each element x_i in the solution vector x (of length n) can be given by the following equation:

$$x_i = (b_i - \sum_{j=n}^{i+1} x_j U_{i,j}) \; / \; U_{i,i}$$

where we assume for convenience that the submatrices are of unit size (and are thus represented simply as scalar quantities). Given this equation for each element, it is easy to construct the solution vector in ParAlfl, as shown in Figure 5.

This problem, as it is, has plenty of parallelism. To see this, look at Figure 6, a dataflow graph showing the data dependencies when $n=5$. Clearly, once an element of the solution is computed, all of the backsubstitutions of it can be done in parallel; that is, each of the horizontal "steps" in Figure 6 can be executed. This parallelism derives solely from the data dependencies inherent in the problem, and is mirrored faithfully in the ParAlfl code. Indeed, if we have n processors sharing a common memory, we can annotate the program in Figure 5 very simply:

```
{ result xvect;
  xvect = = mka(n,x);
  x(i) = = { ... } $on i
}
```

where "..." denotes the same expression used in Figure 5 for $x(i)$. (Recall that the elements of an array are computed in parallel, and thus do not require eager annotations.)

But let us consider topologies that are more interesting. Consider, for example, a *ring* of n processors. Although the topology of a ring is simple, its limited capacity for interprocessor communication makes it difficult to use effectively, and it is thus a challenge for algorithm designers. We will assume that the processors are labeled consecutively around the ring from "1" to "n," and that the ith row of U and ith element of b are on processor i. We wish the solution vector x to be distributed in the same way.

We should first note that the annotated program two paragraphs above would run perfectly well on such a topology, especially with the given distribution of data. The only data movement, in fact, would be that of each submatrix x_i for use on each processor j, $j>i$. This data movement would be done transparently by the underlying operating system, and in this case the program would probably perform adequately.

```
{ result xvect;
  xvect = =mka(n,x);
  x(i) = = { result (b[i]-sum(n,0)) / U[i,i];
             sum(j,acc) = =if j<i+1 then acc
                          else sum(j-1,acc+xpipe[i][n-j+1]*U[i,j])
           } $on i;
  xpipe = =mka(n,xfn);
  xfn(i) = = { result mka(n-i+1,xlocal);
               xlocal(j) = =if j=n-i+1 then xvect[i]
                           else xpipe[i+1][j] } $on i
}
```

Figure 8. A program for pipelining x_i around a ring of processors.

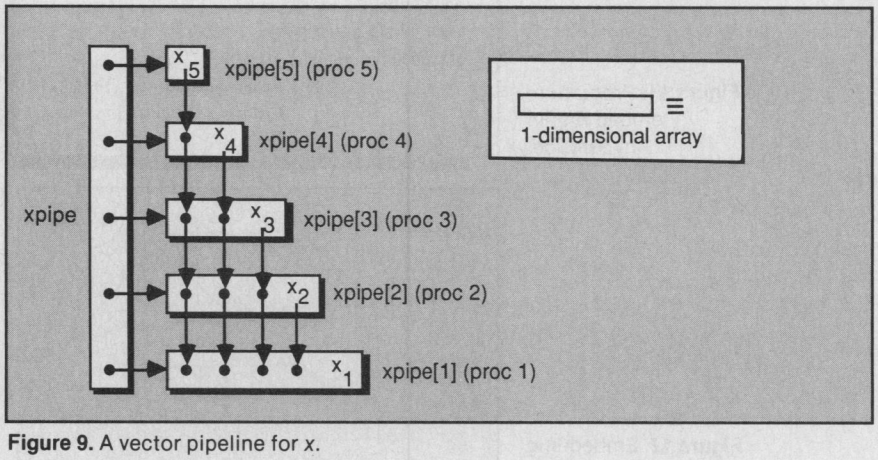

Figure 9. A vector pipeline for x.

Yet in our dual role of programmer and algorithm designer we may have a particular routing strategy that is provably good and that we wish to express explicitly in the program. For example, one efficient strategy is to "pipeline" the x_i around the ring as they are generated. That is, the element x_i is passed to processor *i-1*, used there, passed to processor *i-2*, used there, and so on, as shown graphically in Figure 7. There are several ways to accomplish this effect in the program, and we shall explore two of them.

The first requires the least change to the existing program, and is based on shifting the data by creating a partial copy of the solution vector on each processor, as shown in Figure 8. Note that the first four lines of this program are essentially the same as those given earlier. Figure 9 shows the construction of *xpipe*—note the correspondence between this diagram and the one in Figure 7.

The second way to express the pipelining of data is to interpret the algorithm from the outset as a network of dynamic processes rather than as a static set of vectors and arrays. In particular, we can conjure up the following description of a process running on processor i:

> "Process i takes as input a stream of values x_n, x_{n-1}, ..., x_{i+1}. It passes this stream of values to process *i-1* while back-substituting each value into b_i. When the end of the stream is reached, it computes x_i and adds this to the end of the stream being passed to process *i-1*."

Assuming the same distribution of U and b used earlier, we can represent this process description in ParAlfl as shown in Figure 10. Note that *xi* is annotated for eager evaluation, to override the lazy evaluation of lists. Also note the correspondence between this program and the last. The main

```
{ result process(n,[]);          (Comment: begin on processor n with empty stream)
  process(i,xstr) = =
     { xi = =(b[i] – sum(xstr,n,0))/U[i,i];
       result if i = 1 then addtostr(xstr,#xi)
                       else process(i – 1,addtostr(xstr,#xi));
       addtostr(old,x) = =old^^ [x];
       sum(str,j,acc) = =if str =[] then acc
                              else sum(tl(str),j – 1,acc + hd(str)*U[i,j])
     } $on i
}
```

Figure 10. Program for $Ux = b$ simulating network of processes.

```
ringtocube(i) = =v[i];
v = =mka(n,graycode);
graycode(i) = =if i < 2 then i
                    else { result v[2*mid – i – 1] + mid;
                           mid = =2**log2(i) }
```

Figure 11. Program to embed ring in hypercube.

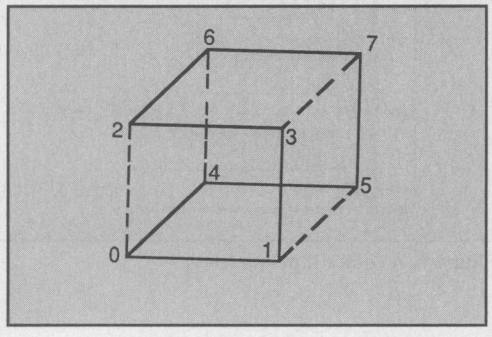

Figure 12. Embedding of ring of size 8 into 3-cube.

difference is in the choice of data structure for x—a list is used here, resulting in a recursive structuring of the program, whereas a vector was used previously, resulting in a "flat" program structure. Choices of this kind are in fact typical of any suitably rich programming language, and are equally important in parallel and sequential programming. Different data structures can, of course, be mapped in different ways to machines, but in this example the annotations are essentially the same in both programs.

To carry this example one step further, let us now consider running any of the above ParAlfl programs on a multiprocessor with a hypercube interconnection topology rather than a ring. One way to accomplish this is to simulate a ring in a hypercube by some suitable embedding. Probably the simplest such embedding is the *reflected gray-code*, captured by the ParAlfl functions shown in Figure 11. In that figure, *log2*(i) returns the base-2 logarithm of i, rounded down to the nearest integer (the vector v is used to "cache" values of *graycode*(i)). For example, Figure 12 shows the embedding of a ring of size 8 into a 3-cube.

If we then replace the previous annotations "... *$on i*" with "... *$on ringtocube*(i)," we arrive at the desired embedding. Note that the code for the algorithm itself did not change at all, just the annotations. Of course, a more efficient algorithm for the hypercube might exist or the initial data distribution might be different, and both cases would naturally require recoding of the main functions.

When viewed in the broad scope of software development methodologies, the use of para-functional programming suggests the following 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 and facilitating the debugging process.
2. Once the program has been written, it is 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, the program is refined by affixing *annotations* that provide more subtle control over the evaluation process. These annotations can be added without affecting the program's functional behavior.

There are two aspects of this methodology that I 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 that carry out complex communication and require complex synchronization. Together with the clean, high-level programming style afforded by functional languages, these two aspects promise to yield a simple and effective programming methodology for multiprocessor computing systems.

Extensions and implementation issues. In this article I have presented only the fundamental ideas behind para-functional programming. Work continues on several advanced features and alternative annotations that provide even more expressive power. These include: (1) annotations that reference other operational aspects of a processor, such as processing load; (2) mappings to operating system resources, such as disks and I/O devices; (3) introduction of nondeterministic primitives where needed; and (4) annotations to control memory usage. The latter two features are especially important, since they allow one to overcome two traditional objec-

tions to programming in the functional style: the inability to deal with the nondeterminism that is prevalent, for example, in an operating system, and inefficiency in handling large data structures. Space limitations preclude me from delving into such issues, but the reader can find additional details in Hudak and Smith[10] and Hudak.[11]

In addition, by concentrating in this article on how to *express* parallel computation, I have left unanswered many questions about how one can *implement* a para-functional programming language. 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. At Yale a virtual parallel graph reducer called *Alfalfa* is currently being implemented on two commercial hypercube architectures: an Intel iPSC and an NCube hypercube. This graph-reduction engine will be able to support both implicit (dynamic) and explicit (annotated) task allocation. The only difficult language feature to support efficiently in para-functional programming is a mechanism for referencing elements in a distributed array; in most cases this is quite easy, but in certain cases it can be difficult. Although good progress has been made in this area, the work is too premature to report here.

Related work. The work that is most similar in spirit to that presented in this article is E. Shapiro's systolic programming in Concurrent Prolog[12]; the mapping semantics of systolic programming was derived from earlier work on "turtle programs" in Logo. Other related efforts include those of R. M. Keller and G. Lindstrom,[13] who, independent of our research at Yale and in the context of functional databases, suggest the use of annotations similar to mapped expressions; and F. W. Burton's[14] annotations to the lambda calculus to provide control over lazy, eager, and parallel execution. A more recent effort is that of N. S. Sridharan,[15] who suggests a "semi-applicative" programming style to control evaluation order. All in all, these efforts contribute to what I think is a powerful programming paradigm in which operational and functional behavior can coexist with little adverse interaction. □

Acknowledgments

I am especially indebted to Lauren Smith (now at Los Alamos National Laboratory), to whom most of these ideas were first presented for critical review. Her efforts at applying the ideas to real problems were especially useful. A special thanks is extended to Yale's Research Center for Scientific Computation (under the direction of Martin Schultz), which provided the motivation for much of this work. The research also benefited from useful discussions with Jonathan Young and Adrienne Bloss at Yale, Bob Keller at the University of Utah, and Joe Fasel and Elizabeth Williams at Los Alamos National Laboratory. Finally, I wish to thank the three anonymous reviewers for *Computer* magazine who worked with my article, as well as Assistant Editor Louise Anderson; their comments helped improve the presentation.

This research was supported in part by NSF Grants DCR-8403304 and DCR-8451415, and a Faculty Development Award from IBM.

References

1. J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM,* Vol. 21, No. 8, Aug. 1978, pp. 613-641.
2. P. Henderson, *Functional Programming: Application and Implementation,* Prentice-Hall, Englewood Cliffs, N. J., 1980.
3. R. M. Keller and F. C. H. Lin, "Simulated Performance of a Reduction-based Multiprocessor," *Computer,* Vol. 17, No. 7, July 1984, pp. 70-82.
4. P. Hudak and B. Goldberg, "Distributed Execution of Functional Programs Using Serial Combinators," *Proc. 1985 Int'l Conf. on Parallel Processing,* Aug. 1985, pp. 831-839; also appeared in *IEEE Trans. Computers,* Vol. C-34, No. 10, Oct. 1985, pp. 881-891.
5. P. Hudak, "ALFL Reference Manual and Programmer's Guide," research report YALEU/DCS/RR-322, 2nd ed., Oct. 1984, Yale University, Dept. of Computer Science, Box 2158 Yale Station, New Haven, CT 06520.
6. D. A. Turner, "Miranda: A Non-strict Functional Language with Polymorphic Types," *Functional Programming Languages and Computer Architecture,* Sept. 1985, Springer-Verlag, New York, pp. 1-16.
7. R. M. Keller, tech. report, "FEL Programmer's Guide," No. 7, Mar. 1982, University of Utah, Dept. of Computer Science, Merrill Engineering Bldg., Salt Lake City, UT 84112.
8. L. Augustsonn, "A Compiler for Lazy ML," *ACM Symp. on LISP and Functional Programming,* Aug. 1984, pp. 218-227.
9. J. Darlington, P. Henderson, and D. A. Turner, *Functional Programming and Its Applications,* Cambridge University Press, Cambridge, UK, 1982.
10. P. Hudak and L. Smith, "Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems," *12th ACM Symp. on Principles of Programming Languages,* Jan. 1986, pp. 243-254.
11. P. Hudak, "Exploring Para-Functional Programming," research report YALEU/DCS/RR-467, Apr. 1986, Yale University, Dept. of Computer Science, Box 2158 Yale Station, New Haven, CT 06520.
12. E. Shapiro, "Systolic Programming: A Paradigm of Parallel Processing," tech. report CS84-21, Aug. 1984, The Weizmann Institute of Science, Dept. of Applied Mathematics; appeared in *Proc. Int'l Conf. on Fifth-Generation Computer Systems,* Nov. 6-9, 1984, pp. 458-470.

13. R. M. Keller and G. Lindstrom, "Approaching Distributed Database Implementations Through Functional Programming Concepts," *Int'l Conf. on Distributed Systems,* May 1985.
14. F. W. Burton, "Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs," *ACM Trans. on Programming Languages and Systems,* Vol. 6, No. 2, Apr. 1984.
15. N. S. Sridharan, tech. report, "Semi-applicative Programming: An Example," Nov. 1985, BBN Laboratories, Cambridge, Mass.

Paul Hudak received his BS degree in electrical engineering from Vanderbilt University, Nashville, Tenn., in 1973; his MS degree in computer science from the Massachusetts Institute of Technology, Cambridge, Mass., in 1974; and his PhD degree in computer science from the University of Utah, Salt Lake City, Utah, in 1982. From 1974 to 1979 he was a member of the technical staff at Watkins-Johnson Co., Gaithersburg, Md.

Hudak is currently an associate professor in the Programming Languages and Systems Group in the Dept. of Computer Science at Yale University, New Haven, Conn., a position he has held since 1982. His primary research interests are functional and logic programming, parallel computing, and semantic program analysis.

He is a recipient of an IBM Faculty Development Award (1984-85), and an NSF Presidential Young Investigator Award (1985).

Readers may write to Paul Hudak at Yale University, Dept. of Computer Science, Box 2158 Yale Station, New Haven, CT 06520.