A Simple Program Transformation for Parallelism

Saumya Debray
Mudita Jain
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
`{debray, jainm}@cs.arizona.edu`

Abstract

Most of the research, to date, on optimizing program transformations for declarative languages has focused on sequential execution strategies. In this paper, we consider a class of commonly encountered computations whose "natural" specification is essentially sequential, and show how algebraic properties of the operators involved can be used to transform them into divide-and-conquer programs that are considerably more efficient, both in theory and in practice, on parallel machines.

1 Introduction

Among the advantages claimed for declarative programming languages are that (*i*) programs written in such languages are relatively easy to reason about, which makes it possible to automatically transform simple declarative specifications into efficiently executable code; and (*ii*) it is easy to exploit parallelism in programs written in such languages. Nevertheless, most of the research on transformation of programs in high level languages has focused, to date, on execution strategies that are—explicitly or implicitly—sequential (see, for example, [1, 2, 4, 18]). As a result, it is not at all certain that the application of such techniques leads to programs that run faster on parallel implementations: performance improvements, if any, are purely incidental.

In this paper, we focus on transforming declarative programs for efficient parallel execution. We consider a class of commonly encountered programs whose "natural" specification is essentially sequential. We use algebraic properties of operators to transform such programs into a divide-and-conquer form that is considerably more efficient on parallel machines, both in theory and in practice, but that do not appear to be "natural" specifications of the computation.

The computations we consider are accumulative computations involving associative operators. The idea behind these accumulative computations is as follows: starting in some "state" $\bar{x}$, repeatedly perform the following computation as long as the state satisfies some condition $a(\bar{x})$: update the accumulator $\bar{y}$, using an associative operator $\oplus$, to accumulate some function[1] $d(\bar{x})$ of the current state; and update the state $\bar{x}$ so that the new state is give by some function $e(\bar{x})$ of the current state. Finally, when this accumulative process stops, update the accumulator a final

[1]We use the word "function" here primarily to help the reader's intuition: as we will see, our transformation does not require determinacy.

time with some function $b(\bar{x})$ of the final state. Accumulative computations of this kind, involving associative operators, are commonly encountered in a wide variety of contexts, to the point that some authors have proposed augmenting programming languages with constructs designed specifically to handle such computations [13, 14].

It is easy to show that this computation can be expressed using a procedure p defined as follows (to keep the program simple, the accumulator has not been made explicit here: it can easily be transformed to a tail-recursive procedure that manipulates an accumulator explicitly: see, for example, [4]). Here, $\bar{x}$ is a tuple of "input" arguments and $\bar{y}$ a tuple of "output" arguments:

$$\begin{array}{lll} p(\bar{x}, \bar{y}) & :- & a(\bar{x}),\ e(\bar{x}, \bar{x}_1),\ p(\bar{x}_1, \bar{y}_1),\ d(\bar{x}, \bar{u})\ y = \bar{u} \oplus \bar{y}_1. \\ p(\bar{x}, \bar{y}) & :- & \neg a(\bar{x}), b(\bar{x}, \bar{y}). \end{array}$$

For convenience in the discussion that follows, we refer to such procedures as *associative-accumulative procedures.* A significant aspect of this procedure is that it is essentially sequential: in order to compute the result at one level of recursion, it is necessary to wait until the rest of the recursive computation has been completed (a similar remark applies to the tail-recursive version that explicitly manipulates an accumulator, since an iteration cannot proceed until the accumulator has been updated by the previous iteration).[2] Notice that we do not make any assumptions about whether or not the procedures d and b are deterministic (i.e., compute functions).

A Note on Notation: *For notational convenience in the remainder of this paper, we depart from the "standard syntax" of logic programming languages in two minor ways. First, because the directionality of the procedures b, d, and e above, and the distinction between their input and output arguments, is crucial for our transformation, it is convenient to think of them as "functions" that return values (though they may be nondeterministic and may return multiple values) and write them in a functional notation. Second, the two clauses given above for the procedure p—one that checks whether recursion should continue, the other whether it should stop—are necessarily mutually exclusive, and we find it useful to emphasize this by writing them using a "guarded" syntax. Thus, we write the definition given above for p as follows:*

$$\begin{array}{lll} p(\bar{x}, \bar{y}) & :- & a(\bar{x})\ [\!]\ p(e(\bar{x}), \bar{y}_1),\ \bar{y} = d(\bar{x}) \oplus \bar{y}_1. \\ p(\bar{x}, \bar{y}) & :- & \neg a(\bar{x})\ [\!]\ \bar{y} = b(\bar{x}). \end{array}$$

We will use such notation often in the remainder of the paper, with the understanding that this can be easily expanded into the original relational notation wherever necessary. However, code for example programs will be given in syntax close to that of Prolog.

2 The Basic Transformation

To motivate the transformation, consider the evaluation of a goal $p(\bar{t}, \bar{y})$ given the definition

[2] It may sometimes be possible to extract a limited amount of parallelism by executing the different invocations of $d(\ldots)$ in parallel. All such opportunities for parallelism are retained by our transformation, which additionally restructures the computation to expose a great deal more parallelism.

$$\begin{array}{lll} p(\bar{x}, \bar{y}) & :- & a(\bar{x}) \; [\!] \; p(e(\bar{x}), \bar{y}_1), \; \bar{y} = d(\bar{x}) \oplus \bar{y}_1. \\ p(\bar{x}, \bar{y}) & :- & \neg a(\bar{x}) \; [\!] \; \bar{y} = b(\bar{x}). \end{array}$$

Suppose the depth of recursion for the input $\bar{t}$ is n, then the result(s) of the evaluation is given by

$$d(\bar{t}) \oplus d(e(\bar{t})) \oplus \cdots \oplus d(e^{n-1}(\bar{t})) \oplus b(e^n(\bar{t})) \tag{1}$$

where $e^j(\bar{t})$ denotes the result of j applications of e to $\bar{t}$. The crucial insight behind the transformation is that if we can (i) determine the depth of recursion n, and (ii) enumerate the exponents of e in the expression (1)—that is, the sequence $1 \ldots n$—efficiently in parallel, then the subexpressions $e^i(\bar{t})$ can also be evaluated in parallel. If this can be done efficiently,[3] so can the entire expression (1). In Section 2.1, we discuss conditions under which the depth of recursion of an associative-accumulative function can be determined. In Section 2.2, we focus on the second issue, namely, the efficient enumeration of the sequence $1, \ldots, n$ given the depth of recursion n, and its utilization in transforming the program into a divide-and-conquer form that can be evaluated efficiently in parallel.

2.1 Calculating the Depth of Recursion

In general, the depth of recursion of a recursive procedure depends on the "size" of its input, where the size of a term is given by a function, called a *norm*, that maps terms to natural numbers. Techniques for automatically inferring the norm appropriate for a particular procedure, based on the types of its input arguments, have been investigated by Decorte *et al.* [6]. For our purposes, we assume that input and output arguments for procedures have been identified via mode analysis; that the types for these arguments have been determined; and that the appropriate norms for the input arguments of procedures have been given, either via programmer annotations, or through dataflow analysis as in [6]. Given a procedure p in a program P, let *in_type*(p) denote the type of the input arguments of p. We focus on a particular class of procedures where the size of the input arguments decreases by a fixed amount at each recursive call, and where the recursion stops once the argument size has reached some minimum value.

Definition 2.1 An associative-accumulative procedure

$$\begin{array}{lll} p(\bar{x}, \bar{y}) & :- & a(\bar{x}) \; [\!] \; p(e(\bar{x}), \bar{y}_1), \; \bar{y} = d(\bar{x}) \oplus \bar{y}_1. \\ p(\bar{x}, \bar{y}) & :- & \neg a(\bar{x}) \; [\!] \; \bar{y} = b(\bar{x}). \end{array}$$

is *size-driven* with respect to a norm $| \cdot |$ if and only if there exist integers $M, N \geq 0$ such that for every $\bar{t} \in$ *in_type*(p), (i) $| \bar{t} | - | e(\bar{t}) | = M$; and (ii) $| \bar{t} | \leq N$ implies $\neg a(\bar{t})$. ∎

Note that e is not required to be a function: it may be nondeterministic, and compute multiple solutions, as long as for all (appropriate) inputs $\bar{t}$ it is the case that for each solution computed for $e(\bar{t})$, the value of $| e(\bar{t}) |$ is the same. This is the case, for example, where given a set S, $e(S)$ yields a subset of S obtained by nondeterministically removing an element from it.

Given a size-driven associative-accumulative procedure where the argument size decreases by M at each recursive call, and where the recursion stops once the

[3]This may not always be possible, e.g., if e computes the tail of a list, the evaluation of $e^i(L)$ requires a sequential traversal of i elements of L.

Input: A size-driven associative-accumulative procedure

$$\begin{aligned} p(\bar{x},\bar{y}) &\;:-\; a(\bar{x}) \;[\!]\; p(e(\bar{x}),\bar{y}_1),\ \bar{y} = d(\bar{x}) \oplus \bar{y}_1. \\ p(\bar{x},\bar{y}) &\;:-\; \neg a(\bar{x}) \;[\!]\; b(\bar{x},y). \end{aligned}$$

where $\oplus$ is associative.

Output: The following procedures, where $\rho(\bar{x})$ is the recursion depth of p for input arguments $\bar{x}$, and q is a new procedure not appearing in the original program:

$$\begin{aligned} p(\bar{x},\bar{y}) &\;:-\; a(\bar{x}) \;[\!]\; q(\bar{x},1,\rho(\bar{x}),\bar{y}_1),\ \bar{y} = \bar{y}_1 \oplus b(e^{\rho(\bar{x})}(\bar{x})). \\ p(\bar{x},\bar{y}) &\;:-\; \neg a(\bar{x}) \;[\!]\; \bar{y} = b(\bar{x}). \end{aligned}$$

$$\begin{aligned} q(\bar{x},m,n,\bar{y}) &\;:-\; m = n \;[\!]\; \bar{y} = d(e^{m-1}(\bar{x})). \\ q(\bar{x},m,n,\bar{y}) &\;:-\; m < n \;[\!] \\ &\quad m' = \lfloor (m+n)/2 \rfloor,\ q(\bar{x},m,m',\bar{y}_1),\ q(\bar{x},m'+1,n,\bar{y}_2),\ \bar{y} = \bar{y}_1 \oplus \bar{y}_2. \end{aligned}$$

Figure 1: The Basic Transformation

argument size drops below N, it is straightforward to determine the recursion depth $\rho(n)$ for an argument of size n:

$$\rho(n) = \min\{k \mid n - kM \le N\} = \lceil (n-N)/M \rceil.$$

2.2 Transformation to Divide-and-Conquer Form

For any given k, the enumeration of the sequence $1,\ldots,k$ in $O(\log k)$ parallel steps can be achieved by the following function: *enum*(m,n) enumerates the sequence of integers $m,\ldots,n$ using a straightforward divide-and-conquer algorithm.

$$\begin{aligned} &\mathit{enum}(m,n) = \mathbf{if}\ m = n\ \mathbf{then}\ \langle\ m\ \rangle \\ &\qquad \mathbf{else}\ \mathit{enum}(m,m') :: \mathit{enum}(m'+1,n)\ \mathbf{where}\ m' = \lfloor (m+n)/2 \rfloor \end{aligned}$$

Here, sequences are enclosed in angle brackets $\langle\cdots\rangle$, and '::' denotes concatenation of sequences. This immediately suggests a technique for transforming an associative-accumulative procedure, with associative operator $\oplus$, to a divide-and-conquer form that can be evaluated more efficiently in parallel. The idea is to pass an additional pair of arguments m and n into the transformed procedure to specify the integer interval $[m,n]$. The structure of the transformed procedure closely follows that of the function *enum*, given above: if the interval passed is a singleton, i.e., if $m = n$, the recursion terminates; otherwise, the input interval is subdivided into two parts, each of these processed recursively, and the result combined using $\oplus$. The form of the transformed definition is shown in Figure 1. The transformation is illustrated by the following example:

Example 2.1 The following procedure computes, for any $n \ge 0$, the factorial of n:

```
fact(N, F) :- N > 0  []  fact(N-1, F1), F = N*F1.
fact(0, 1).
```

In this case, $a(n) \equiv n > 0$; b is the constant function `1`; d is the identity function; $e(n) = n - 1$; and the operator $\oplus$ is integer multiplication, `*`. Given an input n, the argument to the recursive call is $n - 1$, so the depth of recursion is n. Finally, since $e(n) = n - 1$, the value of e^k for any $k \geq 0$ is $n - k$. The transformed procedure, therefore, is:

```
fact(N,F)  :- N > 0  ‖  fact_1(N,1,N,F1), F = 1*F1.
fact(0,1).

fact_1(N,K,M,F)  :- K = M  ‖  F = N-(K-1).
fact_1(N,K,M,F)  :- K < M  ‖
     K1 = ⌊(K+M)/2⌋, fact_1(N,K,K1,F1), fact_1(N,K1+1,K,F2),
     F = F1*F2.
```

The resulting procedures can easily be simplified so as to improve their performance. First, `1*F1` = `F1`. Second, the second and third arguments of `fact_1`/4 represent the lower and upper bounds of an integer interval, and are necessarily nonnegative integers,[4] so the computation '`K1 = ⌊(K+M)/2⌋`' can be implemented by an integer addition followed by a right shift of 1 bit, which is considerably less expensive. The resulting program is:

```
fact(N,F) :- N > 0  ‖  fact_1(N,1,N,F).
fact(0,1).

fact_1(N,K,M,F) :- K = M  ‖  F = N-(K-1).
fact_1(N,K,M,F) :- K < M  ‖
     K1 = (K+M)>>1, fact_1(N,K,K1,F1), fact_1(N,K1+1,K,F2),
     F = F1*F2.
```

□

In the transformation described above, an integer interval is specified using two arguments that represent the two endpoints of the interval. Of course, any other representation of an interval could have been used: for example, we could have used one endpoint and the length of the interval. The transformations corresponding to such alternate representations are straightforward to derive following the basic approach outlined earlier, and the resulting procedures have essentially the same structure and complexity as those obtained from the transformation described here, so we do not discuss them separately.

The following theorem shows the correctness of the transformation: the proof is omitted due to space constraints.

Theorem 2.1 *Let p be any associative-accumulative procedure, and p' the corresponding procedure obtained by the transformation shown in Figure 1. Then for any input arguments $\bar{t}$, the set of answer substitutions for the goal $p(\bar{t}, \bar{y})$ is the same as that for the goal $p'(\bar{t}, \bar{y})$.* ■

It is not hard to see that any opportunities for AND-parallel execution in the original program are retained in the transformed program. Thus, computations that could

[4]This is inherent in the transformation itself, and does not require any program analysis, so it is quite reasonable to expect a compiler to generate the optimized code shown.

have been executed in parallel in the original program can still be executed in parallel in the transformed program, and any performance benefits accruing from such parallel execution—which is what would be obtained using a parallelizing compiler such as &-Prolog's—are obtained also in the transformed program.

2.3 Optimizations to the Basic Transformation: Granularity Control

An issue that parallel implementations of very high level languages have to contend with, in general, is that of task granularity. Because of the overheads associated with task creation and management, the creation of a large number of fine-grained parallel tasks can, in practice, cause a deterioration in the performance of a system even though these tasks are executed in parallel. On the other hand, a system that is overly conservative in this regard, and creates too few parallel tasks, may fail to exploit the available parallelism effectively, resulting in suboptimal performance. Thus, there is a need to balance the amount of parallelism exploited against the overheads incurred in doing this. The question of task granularity control has been investigated by a number of researchers (see, e.g., [5, 16]).

The transformation described above is easily amenable to granularity control. Most of the computation in the transformed program shown in Figure 1 is carried out by the auxiliary procedure q, which manipulates a (nonempty) integer interval: the recursion terminates if and only if the interval is a singleton, i.e., the upper bound is equal to the lower bound. This procedure can therefore be rewritten as

$$\begin{array}{l} q(\bar{x}, m, n, \bar{y}) \ :- \ n - m \geq 1 \ [\!] \\ \qquad m' = \lfloor (m+n)/2 \rfloor, \ q(\bar{x}, m, m', \bar{y}_1), \ q(\bar{x}, m'+1, n, \bar{y}_2), \ \bar{y} = \bar{y}_1 \oplus \bar{y}_2. \\ q(\bar{x}, m, n, \bar{y}) \ :- \ n - m = 0 \ [\!] \ \bar{y} = d(e^{m-1}(\bar{x})). \end{array}$$

Now suppose that for a given program and implementation, we decide—either from user-supplied information; or from profile information, as in [16]; or using program analysis techniques, as in [5]—that it is worth creating a parallel task for this function only if the interval being processed is of size at least N. Since the interval processed by each recursive call is about half of the input interval, this means that it is worth spawning the recursive calls in parallel only if the input interval is of size $2N$. The transformed function can then be implemented as

$$\begin{array}{l} q(\bar{x}, m, n, \bar{y}) \ :- \ n - m \geq 2N \ [\!] \\ \qquad m' = \lfloor (m+n)/2 \rfloor, \ q(\bar{x}, m, m', \bar{y}_1), \ q(\bar{x}, m'+1, n, \bar{y}_2), \ \bar{y} = \bar{y}_1 \oplus \bar{y}_2. \\ q(\bar{x}, m, n, \bar{y}) \ :- \ n - m < 2N \ [\!] \ q\_seq(\bar{x}, m, n, \bar{y}). \end{array}$$

where *q_seq* has the same behavior as q but processes the interval $m \ldots n$ sequentially. Let ε be the identity for the operator $\oplus$, then this procedure can be defined in the following tail-recursive form that can be efficiently executed sequentially:

$$q\_seq(\bar{x}, m, n, \bar{y}) \ :- \ q\_seq\_1(e^{m-1}(\bar{x}), n-m, \varepsilon, \bar{y}).$$

$$\begin{array}{l} q\_seq\_1(\bar{x}, n, \bar{w}, \bar{y}) \ :- \ n = 0 \ [\!] \ \bar{y} = d(\bar{x}) \oplus \bar{w}. \\ q\_seq\_1(\bar{x}, n, \bar{w}, \bar{y}) \ :- \ n > 0 \ [\!] \ q\_seq\_1(e(\bar{x}), n-1, d(\bar{x}) \oplus \bar{w}, \bar{y}). \end{array}$$

One of the observations from the the granularity control experiments reported in [5] is that in general, keeping track of argument sizes for dynamic granularity control can incur additional runtime overheads that can, in some cases, swamp the performance gains resulting from granularity control and lead to a slowdown in the execution speed of the program. It can be seen, however, that in the case of this transformation, the tests required for granularity control fit smoothly into the transformed code, so that no additional overhead is incurred.

3 Generalizations

3.1 Adaptive Partitioning

The discussion of the transformation so far has assumed implicitly that, when processing an integer interval $[m, n]$ with $m < n$, the appropriate place to split this interval for recursive processing is at $\lfloor (m+n)/2 \rfloor$. This may be a reasonable choice in many cases, but does not appear to be *a priori* necessary for correctness, and other choices of the split point may sometimes be more appropriate. To this end, suppose we use a function $split(m, n)$ to determine where to split an interval $[m, n]$. What properties must such a function satisfy? Assume that, as before, recursion terminates when the interval being processed becomes a singleton. Thus, an interval $[m, n]$ is split only if $m < n$. In this case, correctness requires only that each of the two subintervals resulting from the split be smaller than the interval $[m, n]$. This is satisfied if *split* is contractive:

Definition 3.1 A function $f : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}$ is *contractive* if $m \leq f(m, n) < n$ for all $m, n \in \mathcal{N}$. ■

To see that correctness is preserved if the function *split* is contractive, note that whenever the recursive clause is executed, the size of the integer interval associated with each of the recursive calls is strictly smaller than that of the input interval, and that recursion terminates when the input interval becomes a singleton. This means that termination is preserved. Furthermore, only the non-recursive clause corresponds to a leaf node in the expression tree being evaluated, and such leaf nodes are unaffected by the choice of the split point in the recursive clause. Thus, different choices of the split point affect only the shape of the expression tree evaluated but not its leaves. Since the operator $\oplus$ is assumed to be associative, the value of the expression tree depends only on its leaves. This means that other choices fo the split point based on a contractive function do not affect the result of the computation.

Applications of adaptive partitioning are given in Sections 4.1.2 and 4.2.2.

3.2 Multi-way Divide and Conquer

Intuitively, the transformation discussed above is guided by analogy with a divide and conquer scheme that enumerates a sequence of integers $1, \ldots, n$ by dividing the input interval into two halves, enumerating each one, and concatenating the results. If we pursue this analogy, it is easy to see how our basic transformation can be generalized to multi-way divide-and-conquer, by simply dividing the interval into k sub-intervals of more or less equal size at each stage, for any fixed $k \geq 2$. In this case, since the recursive computation involves division by k, the base case should account for the remainders possible from this, namely, $0, \ldots, k-1$. Thus, recursion should stop if $n - m < k$. The details are straightforward, and due to space constraints we do not present them separately. The following example shows how this works (this is provided more for the reader's amusement than as a serious suggestion for a realization of the factorial function, but the technique can be quite useful for exploiting parallelism in less trivial computations).

Example 3.1 The factorial function defined in Example 2.1 can be transformed to use a 5-way divide-and-conquer algorithm as follows:

```
fact(N, F) :- N > 0  ‖  fact_1(N, 1, N, F).
fact(0, 1).

fact_1(N, K, M, F) :- M-K < 5  ‖  fact_2(M-K, N-(K-1), 1, F).
fact_1(N, K, M, F) :- M-K >= 5  ‖
     M1 = floor((M+K)/5), fact_1(N, K, M1, F1),
     M2 = floor(2(M+K)/5), fact_1(N, M1+1, M2, F2),
     M3 = floor(3(M+K)/5), fact_1(N, M2+1, M3, F3),
     M4 = floor(4(M+K)/5), fact_1(N, M3+1, M4, F4),
     fact_1(N, M4+1, N, F5),
     F = F1*F2*F3*F4*F5.

fact_2(0, S, P, F) :- F = S*P.
fact_2(N, S, P, F) :- N > 0  ‖  fact_2(N-1, S-1, S*P, F).
```

□

The correctness argument for the transformation to multi-way divide-and-conquer follows the same lines as that for the basic transformation, and is not repeated here.

4 Applications to Program Synthesis

In this section we show how the ideas discussed earlier can be applied (with user guidance) to transform simple but inefficient programs into more complex but efficient (not necessarily parallel) algorithms.

4.1 Searching

To motivate our approach, we start with a very simple search procedure: given an array of integers `A` and an integer `X`, the call `search(L, X, Found)` binds `Found` to `1` if `X` occurs in the list `L`, and to `0` if it does not. The procedure `search/3` can be defined as follows:

```
search(A,X,Found)  :- search(A,X,1,Found).

search(A,X,I,F)  :- I > size(A)  ‖  F = 0.
search(A,X,I,F)  :- I ≤ size(A)  ‖
   found(A,I,X,F1), search(A,X,I+1,F2), F = F1 ∨ F2.
```

In this case, of course, the predicate `found/4` is quite trivial—`found(A, I, X, F)` binds `F` to `1` if the $\mathtt{I}^{th}$ element of `A` is `X`, and to `0` otherwise—and we could have written the definition for `search/4` much more simply. As we will see, however, the formulation given above is useful in that it allows us to express various other search problems in a very similar way. For example, if instead of a "flat" array we had arbitrarily nested arrays, and wanted to recursively traverse these nested arrays when looking for an element, we could reuse the definition given above simply by redefining `found/4` appropriately. We hope that the reader will agree that, even if we ignore any benefits accruing from such flexibility, the definition given above has the merits of being simple and "obviously correct." Notice also that while the definition given above indicates only whether a given value occurs in a given array, it is easy to modify it so that it computes a position in the array where the value occurs, if there is such a position, or `0` if it does not occur in the array: the procedure `found`/4 has to be modified in the obvious way, and the operator $\vee$ has to be replaced by *max*.

The procedure `search`/4 defined above is easy to recognize as an instance of a size-driven associative-accumulative computation. Applying the basic transformation yields, with very little effort, a program that can search an array of n elements in parallel in $O(\log\ n)$ time: we omit the code for this due to space constraints.

4.1.1 Transformation to Binary Search

The divide-and-conquer search algorithm obtained from the basic transformation above can be improved with more knowledge about the input array. Suppose we have a predicate `must_be_in`/4 such that, given an array `A`, and indices `I` and `J`, `must_be_in(X, A, I, J)` is true if and only if `X` must occur between within the subarray of `A` spanned by the interval `I`, ..., `J` if it occurs as an element of `A` at all. The procedure `search`/6 can then be rewritten as:

```
search_1(A,X,I,K,M,F)  :- K = M  ‖  found(A,I,X,F).
search_1(A,X,I,K,M,F)  :- K < M  ‖
     K1 = floor((K+M)/2),
     (must_be_in(X,A,K,K1) →
          search_1(A,X,I,K,K1,F) ; search_1(A,X,I,K1+1,M,F)).
```

In particular, suppose we know that `A` is sorted, so that `must_be_in`/4 can be defined as follows, where `A[J]` denotes the J^{th} element of `A`:

```
must_be_in(X,A,I,J)  :- X ≤ A[J].
```

Substituting this back into the definiton of `search`/6 yields an implementation of binary search. The resulting program can be further optimized in various ways. Since the auxiliary predicate `search`/4 is called from exactly one place, and can be "in-lined" away. The techniques of Ramakrishnan *et al.* [15] can be used to detect that the third argument of `search`/6 is never defined or used, and can therefore be discarded. This yields the program:

```
search(A,X,Found)  :- size(A) < 1 ‖ Found = 0.
search(A,X,Found)  :- size(A) ≥ 1 ‖ search_1(A,X,1,size(A),Found).

search_1(A,X,K,M,F)  :- K = M  ‖  found(A,K-1,X,F).
search_1(A,X,K,M,F)  :- K < M  ‖
   K1 = floor((K+M)/2),
   (X ≤ A[K1] → search_1(A,X,K,K1,F) ; search_1(A,X,K1+1,M,F)).
```

4.1.2 Transformation to Interpolation Search

The transformation to binary search always splits the interval under consideration at the middle. However, if we have additional information about the distribution of values in an interval, it makes sense to use this information to focus on a "neighborhood" where the value must lie, instead of blindly dividing the interval at its midpoint. Such a search procedure is called an "interpolation search," and is very efficient for inputs where the values are more or less uniformly distributed [11].

Using adaptive partitioning, as discussed in Section 3.1, it is trivial to transform the binary search program obtained above to an interpolation search. Suppose we are searching an array A for a value x. The split function for this is defined as follows (see [11]):

$$\mathit{split}(A, x, m, n) = \lfloor m + (x - A[m])(n - m)/(A[n] - A[m]) \rfloor.$$

It is easy to show that this function is contractive, whence adaptive partitioning based on this function is correct.

4.1.3 Searching with Non-uniform Distributions

The interpolation search described in the previous section works well if the input values are more or less uniformly distributed. However, the adaptive partitioning technique used for the transformation to interpolation search program can easily be used to handle non-uniform distributions as well, as long as the distribution is known, or can be approximated, beforehand. The essential idea, in this case, is that when processing an interval $[m, n]$, the split point m' should be chosen such that the probability that the value being searched for is in the interval $[m, m']$ is as close as possible to the probability that it is in $[m' + 1, n]$. These probabilities can be determined fairly easily for any given distribution of input values, and the transformation thereafter follows the lines described earlier.

4.2 Finding Roots of Equations

Equation-solving can be formulated as a search problem where, given a function f and an interval $[m, n]$, we search the interval to find a value x_0 such that $f(x_0) = 0$ to within some tolerance ε. A straightforward program to do this is given by the following, where `found(I, Eps, R)` binds `R` to `I` if `I` is a root to within a tolerance of `Eps`, and to ∞ otherwise:

```
solve(I,J,Eps,Root)  :- I ≤ J  ‖
   found(I,Eps,R1), solve(I+Eps,J,Eps,R2), Root = min(R1,R2).
solve(I,J,Eps,Root)  :- I > J  ‖  Root = ∞.
```

Applying our basic transformation to this program yields a straightforward divide-and-conquer program for searching for roots in parallel: again, the code is omitted due to space constraints.

4.2.1 Transformation to the Bisection Method

As in Section 4.1.1, we can improve the program above with more information about the function under consideration: as before, define a predicate `must_be_in` such that `must_be_in(I, J)` is true if f has a root in the interval `[I, J]`. The predicate `solve_1` can then be rewritten as follows:

```
solve_1(I,Eps,M,N,Root)  :- M = N ‖ found(I+(M-1)*Eps,Eps,Root).
solve_1(I,Eps,M,N,Root)  :- M < N ‖
    M1 = floor(M+N)/2,
    (must_be_in(M,M1) →
        solve_1(I,Eps,M,M1,Root) ; solve_1(I,Eps,M1+1,N,Root)).
```

In particular, a continuous function f has a zero in an interval $[m, n]$ if the sign of $f(m)$ is different from the sign of $f(n)$, so **must_be_in**/2 can be defined as:

$$\texttt{must\_be\_in(I,J)} \;:-\; \mathit{sign}(\texttt{f(I)}) \neq \mathit{sign}(\texttt{f(J)}).$$

The resulting program finds roots of equations using what is essentially the bisection method [17].

4.2.2 Transformation to (Modified) Newton-Raphson

Instead of blindly splitting the input interval at the midpoint at each recursive step, we can use adaptive partitioning. A simple approach to doing this would be as follows: given an interval $[m, n]$, approximate the function f under consideration by a straight line tangential to f at the endpoint n, determine where this line crosses the x-axis (i.e., becomes 0), and use this point to split the input interval. The resulting split function is given by the following (see, for example, [17]), where f' denotes the first derivative of f:

$$\mathit{split}(m, n) = n - f(n)/f'(n).$$

If we were to use this split function, the resulting program would use the Newton-Raphson method. However, the *split* function so defined is not contractive, and so does not satisfy our criteria for correctness. There are two problems: if $f'(n)$ is sufficiently small, the value of $n - f(n)/f'(n)$ may be smaller than m; and if $f(n)/f'(n)$ is negative, the value of $n - f(n)/f'(n)$ will be greater than n; in either case, the requirements for contractive functions are violated. An obvious fix is to split at the midpoint of the interval if this happens. The resulting split function is:

$$\begin{aligned}\mathit{split}(m, n) = {} & \textbf{if } m \geq n - f(n)/f'(n) \textbf{ or } f(n)/f'(n) < 0 \textbf{ then } \lfloor (m+n)/2 \rfloor \\ & \textbf{else } n - f(n)/f'(n).\end{aligned}$$

The resulting program is:

```
solve_1(I,Eps,M,N,Root)  :- M = N [] found(I+(M-1)*Eps,Eps,Root).
solve_1(I,Eps,M,N,Root)  :- M < N []
    M1 = split(M,N),
    (must_be_in(M,M1) ->
        solve_1(I,Eps,M,M1,Root) ; solve_1(I,Eps,M1+1,N,Root)).
```

This program uses a nontrivial, and better-behaved, modification of the Newton-Raphson method very similar to one described by Dekker [7].

5 Performance

While the divide-and-conquer programs resulting from our transformation may uncover more parallelism than the original program, it is not immediately obvious whether they are, in practice, "better" than either the original programs, or the programs one might get using existing parallelizing compilers such as that of &-Prolog [9]. The reason for this is that the transformation introduces some additional costs—two additional arguments have to be tested, manipulated and passed around in the transformed program—and it is not obvious that these costs are adequately

Program	Version	No. of Processors						
		1	2	3	5	10	15	20
dp(255)	TR	16.5	16.5	**16.5**	16.5	16.5	16.5	16.5
	&-P	16.5	16.5	**16.5**	16.5	16.5	16.5	16.5
	D&C	35.0	17.6	**11.2**	7.4	4.0	3.0	2.5
e(100)	TR	1434.9	**1434.9**	1434.9	1434.9	1434.9	1434.9	1434.9
	&-P	1522.7	**821.3**	588.4	400.0	257.5	213.9	186.1
	D&C	1498.9	**750.6**	502.3	306.0	165.8	122.6	100.2

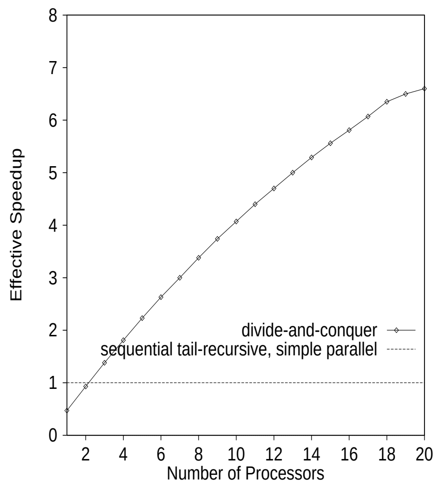
Table 1: Summary of Performance Numbers (in milliseconds)

offset by the benefits of increased parallelism. In order for the transformation to be of practical interest, it is not enough for it to uncover more parallelism than the original program, or to obtain better speedups: it must be shown that the actual parallel execution speed of the transformed program surpasses that of the original program using a reasonable number of processors.

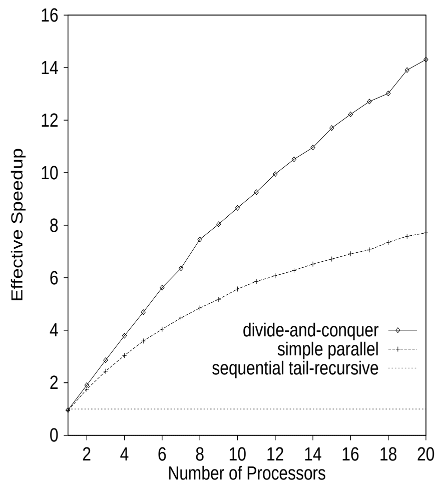
In this section, we give experimental results for two small benchmark programs. The first program, `dp(255)`, computes the dot product of two vectors of length 255; the second, `e(100)`, computes the value of the constant $e = \sum_{i \geq 0} \frac{1}{i!} = 2.71828\ldots$by summing the first 100 terms of the series. Three versions of each program were tested: a straightforward tail-recursive program (TR), the parallel program obtained using the &-Prolog compiler on the tail-recursive program (&-P), and the divide-and-conquer program obtained using our transformation (D&C). The numbers reported were obtained by first running each program under &-Prolog [9] on a Sparcstation-1, and then using the IDRA tool [8] to compute ideal speedups under &-Prolog for more than one processor. In each case, the tail-recursive program was entirely sequential and showed no speedups. The &-Prolog compiler found no parallelism in the `dp(255)` program—it deemed the arithmetic computations too lightweight to be worth doing in parallel—but parallelized `e(100)` to compute factorials in parallel. In the divide-and-conquer version, the recursive calls were executed in parallel.

The results of our experiments are summarized in Table 1: here, boldface entries indicate the "break-even point" for the transformed programs, i.e., the point at which the transformed program is faster than both the tail-recursive sequential implementation and the parallel program obtained from the original program using a parallelizing compiler such as the &-Prolog compiler. Effective speedup curves are given in Figure 2. The numbers indicate that the uniprocessor performance penalty incurred by the transformed program is not very large, even for relatively simple programs, and that the parallel performance of the transformed program quickly surpasses that of both the original program and that obtained using the parallelizing compiler: the break-even point is 3 processors for the `dp(255)` benchmark, and 2 processors for `e(100)`. Moreover, as the number of processors is increased, the divide-and-conquer programs exhibit significantly better speedups than the programs obtained using the parallelizing compiler, and as a result widens the performance gap between the two (the tail-recursive versions are essentially sequential and show no speedups).

In a significant sense, these benchmarks are interesting precisely because they are small: they represent the "worst case" for our transformation. The computa-



(a) `dp(255)`



(b) `e(100)`

Figure 2: Speedup Curves

tion performed at each level of recursion is fairly small, and because of this the performance of these programs is much more sensitive to the additional costs introduced by the transformation, such as parameter passing, comparison operations, and procedure calls, than would be the case if each iteration involved more substantial computations. Nevertheless, the performance of the programs obtained using our transformation is quite good. This suggests that for larger programs, where the computations involved are more "heavyweight" and the additional costs incurred relatively less significant than for smaller programs, the performance can be expected to be correspondingly better. (Unfortunately, practical limitations of the current version of IDRA prevented us from studying significantly larger programs: for example, using our transformation to multiply two 64×64 matrices produced so many parallel tasks that the trace file generated was over 20 MB in size, took over 3 hours to read in, and required too much resources during processing.)

6 Related Work

Conceptually, our ideas are closely related to parallel prefix computation [10]. While parallel prefix algorithms are generally formulated in terms of computing prefixes, in parallel, of some given string, we address the slightly different problem of starting with an encoding of a family of strings (the original program) and producing an encoding of a family of trees (the transformed program). The literature on parallel prefix algorithms also does not usually consider issues such as granularity control, multi-way divide-and-conquer, and adaptive partitioning.

The program transformation work that is probably the closest to ours is that of Bush and Gurd [3]. These authors define a transformation scheme for FP programs that is similar in many ways to the basic transformation described in Section 2. However, they appear to consider only a restricted class of functions: those over integers, and those using involving the list operators *map*, *reduce*, and *generate*, and do not consider the possibility of nondeterminism. Our approach, by contrast, is applicable to any domain that comes equipped with an associative operator and a size function, and can deal with nondeterminism without any problems. Moreover, the treatment of Bush and Gurd does not give any indication of how generalizations, such as to multi-way divide-and-conquer, should be realized, or discuss applications to program synthesis. Finally, Bush and Gurd do not give any empirical evidence of the practicality of their transformation.

Also closely related is Millroth's work on compilation of Reform Prolog [12]. The biggest difference between the two is that Millroth's work relies on low-level aspects of the Reform Prolog system, while ours is formulated as a high-level source-to-source transformation. Because of this, our approach can be easily used on a variety of implementations, and also can be used with user guidance to implement fairly nontrivial transformations (for example, it is not clear how something analogous to the transformation discussed in Section 4.2.2 might be accomplished using Millroth's techniques). On the other hand, Millroth's approach does not require assumptions about operator associativity, and therefore may sometimes be able to exploit parallelism in programs where our approach would not. Finally, for the parallel execution of loops, Millroth relies on techniques developed for Fortran-like languages: he does not consider techniques such as multi-way divide-and-conquer or adaptive partitioning, or issues such as granularity control, that we have discussed here (though in principle there is nothing that precludes these ideas from being incorporated into his approach). More importantly, the reliance on Fortran-like loop-parallelization techniques, unlike our approach, leaves unaddressed the question of parallel execution of loops involving nondeterministic computations, since these do not arise in Fortran-like languages and are not considered in standard texts on Fortran implementation [19, 20].

7 Conclusions

While it is generally believed that programs written in high-level programming languages are amenable to manipulation by powerful semantics-based tools for transformation to more efficient forms on the one hand, and to parallel execution on the other, most of the work on program transformation appears to have focused on sequential execution models. In this paper, we consider a class of computations that occur frequently in practice, and whose "natural" specification is essentially sequential. We describe a simple transformation scheme for such programs that allows them to be executed efficiently in parallel, describe a number of generalizations to the basic transformation, and show how the transformation can also be applied to the derivation of efficient sequential programs starting from simple and obviously correct, but possibly inefficient, programs. Finally, we present simulation results that indicate that the runtime overhead incurred by programs obtained using our transformation is small, and that these programs are significantly superior to both the original programs, and to the programs that might be obtained using existing parallelising systems.

Acknowledgements: We are grateful to Manuel Hermenegildo and members of the CLIP group at Universidad Politecnica de Madrid, Spain, for their help with the &-Prolog and IDRA systems; and to Evan Tick and the anonymous referees for comments on earlier versions of the paper. This work was supported in part by the National Science Foundation under grant CCR-9123520.

References

[1] J. Arsac and Y. Kodratoff, "Some Techniques for Recursion Removal from Recursive Functions", *ACM TOPLAS* vol. 4 no. 2, Apr. 1982, pp. 295-322.

[2] R. M. Burstall and J. Darlington, "A Transformation System for Developing

Recursive Programs", *JACM* vol. 24 no. 1, pp. 44-67, Jan. 1977.

[3] V. J. Bush and J. R. Gurd, "Transforming Recursive Programs for Execution on Parallel Machines", *Proc. Functional Programming Languages and Computer Architecture*, Nancy, France, Sept. 1985, pp. 350–367.

[4] S. K. Debray, "Optimizing Almost-Tail-Recursive Prolog Programs", *Proc. Functional Programming Languages and Computer Architecture*, Nancy, France, Sept. 1985.

[5] S. K. Debray, N. Lin and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 174–188.

[6] S. Decorte, D. De Schreye, and M. Fabris, "Automatic Inference of Norms: a Missing Link in Automatic Termination Proofs", *Proc. 1993 International Symposium on Logic Programming*, Vancouver, B.C., Nov. 1993, pp. 420–436.

[7] T. J. Dekker, "Finding a Zero by means of Successive Linear Interpolation", in *Constructive Aspects of the Fundamental Theorem of Algebra*, eds. B. Dejon and P. Henrici, Wiley-Interscience, London, 1969.

[8] M. J. Fernández, M. Carro, and M. Hermenegildo, "IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages", Technical Report FIM26.3/AI/92, Computer Science Faculty, Technical University of Madrid, September 1992.

[9] M. Hermenegildo and K. Greene, "The &-Prolog System: Exploiting Independent And-Parallelism", *New Generation Computing* vol. 9 nos. 3–4, 1991, pp. 233–257.

[10] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufman, 1992.

[11] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989.

[12] H. Millroth, "Reforming Compilation of Logic Programs", *Proc. 1991 International Symposium on Logic Programming*, San Diego, Oct. 1991, pp. 485–499.

[13] R. S. Nikhil, *Id Language Reference Manual*, Computation Structures Group Memo 284-2, Lab. for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1991.

[14] K. Pingali and K. Ekanadham, "Accumulators: New Logic Variable Abstractions for Functional Languages", *Proc. Eighth Conference on Foundations of Software Technology and Theoretical Computer Science*, Pune, India, Dec.. 1988, pp.377–399. Springer-Verlag LNCS vol. 338.

[15] R. Ramakrishnan, C. Beeri, and R. Krishnamurthy, "Optimizing Existential Datalog Queries", *Proc. Seventh ACM Symp. on Principles of Database Systems*, Austin, TX, March 1988, pp. 89–102.

[16] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, 1989.

[17] L. F. Shampline and R. C. Allen, Jr., *Numerical Computing: an Introduction*, W. B. Saunders, 1973.

[18] H. Tamaki and T. Sato, "Unfold/Fold Transformations of Logic Programs", *Proc. Second International Conference on Logic Programming.* Uppsala, Sweden, 1984.

[19] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

[20] H. Zima, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1991.