Work-Efficient Nested Data-Parallelism†

Daniel W. Palmer and Jan F. Prins
Department of Computer Science
University of North Carolina
Chapel Hill NC 27599-3175
{palmerd,prins}@cs.unc.edu

Stephen Westfold
Kestrel Institute
Palo Alto, CA 94304
westfold@kestrel.edu

Abstract

An *apply-to-all* construct is the key mechanism for expressing data-parallelism, but data-parallel programming languages like HPF and C* significantly restrict which operations can appear in the construct. Allowing arbitrary operations substantially simplifies the expression of irregular and nested data-parallel computations. The technique of *flattening nested parallelism* introduced by Blelloch, compiles data-parallel programs with unrestricted apply-to-all constructs into vector operations, and has achieved notable success, particularly with irregular data-parallel programs. However, these programs must be carefully constructed so that flattening them does not lead to suboptimal work complexity due to unnecessary replication in index operations. We present new flattening transformations that generate programs with correct work complexity. Because these transformations may introduce concurrent reads in parallel indexing, we developed a randomized indexing that reduces concurrent reads while maintaining work-efficiency. Experimental results show that the new rules and implementations significantly reduce memory usage and improve performance.

1 Introduction

1.1 Data-parallelism

A notation permits the expression of data-parallelism if it includes aggregate values such as sets or sequences and the ability to independently apply a function to every element of such an aggregate. A familiar example of data-parallelism is the comprehension construct of set theory: $\{ f(x) \mid x \in A \}$ denotes the set of results obtained as the function f is applied to each value drawn from a set A. This expresses parallelism because all applications of f are independent and hence can be performed simultaneously.

Data parallelism is the main source of concurrency in variants of Fortran developed for architecture-independent parallel programming [1,8,10]. Other languages such as C* and Modula-2* also rely on data-parallelism to express concurrency. In all of these languages the aggregates are restricted to rectangular arrays, and the functions that may be applied in parallel are restricted to a fixed set of elementwise operations, reductions, and parallel prefix operations. While these restrictions simplify the generation of efficient code, they also limit the expressiveness of data parallelism.

These limitations appear in attempts to express irregular parallel computations with a restricted data-parallel language. Consider implementing a parallel version of quicksort in any variety of data parallel Fortran. At each recursive step a list is divided into potentially unequal parts. Mapping these irregular sized sub-problems onto a rectangular aggregate wastes space and serializing the computation wastes time. The programmer is faced with the explicit bookkeeping and data-dependent choice of operations required to manually simulate irregular structures with one-dimensional aggregates. This convoluted task clearly illustrates the price, in terms of expressiveness, of restricting the applicability of apply-to-all constructs.

A more natural way to express irregular computations is with nested data-parallelism. Turning back to set theory, we define a set valued function $f(A,p) = \{(p,q) \mid q \in A \text{ and } 1 \le q \le p\}$. Since sets may contain sets as members, when $A = \{1, 2, 3\}$ the expression denotes $\{\{(1,1)\}, \{(2,1), (2,2)\}, \{(3,1), (3,2), (3,3)\}\}$, a set of sets. The parallelism it expresses is *nested*. For each value p of A, $\{ f(A,p) \mid p \in A\}$ specifies an invocation of f that may be executed in parallel. Each f generates a "nested" set of values for q which may in turn be evaluated in parallel. The distinguishing characteristics of nested data-parallelism are that arbitrary functions are applicable in

† This research supported in part by ARPA via ONR Contract N00014-92-C-0182.

parallel over an aggregate and that aggregates are nestable, that is, they can contain aggregates as values.

The expressive utility of nested data-parallelism was recognized long ago in high level sequential programming languages like SETL [16] and APL2 [11]. Parallel execution of nested data-parallelism has been realized by recent languages such as Paralation Lisp [15], NESL [3] and Proteus [7,9]

In Proteus, our high-level, wide-spectrum parallel language, all data-parallelism is expressed using an iterator construct which is analogous to the comprehension construct of set theory. For example, if `A` is a sorted sequence of integers and `S` is an arbitrary sequence of integers, the iterator expression

```
[ x in S: binsearch(A,x)]                    (1.1)
```

specifies that for each binding of `x` to a value from `S`, the corresponding element of the result sequence is the evaluation of `binsearch(A,x)`. In this case `binsearch` is the sequential definition of binary search with signature $Seq(Int) \times Int \rightarrow Int$.

Nested data parallelism is particularly important in specifying work-efficient irregular parallel computations. The sizes of aggregates within a nested aggregate are independent and can therefore be used to represent the different sized sub-problems specified in irregular computations. For example, suppose we alter expression (1.1) to search several different sized lists for the same element. If `D` is a nested sequence with non-uniformly sized sub-sequences, then the expression

```
[ S in D: binsearch(S,c)]                    (1.2)
```

yields the sequence of locations at which `c` appears in each sub-sequence. Since the number of elements in each invocation of binsearch may differ over the different sub-problems, it specifies irregular parallelism and must operate on a ragged data-structure.

1.2 Flattening nested data-parallelism

Nested data-parallelism has been slow to be adopted into parallel programming notations since it has been thought difficult to implement because of its unpredictable and fine-grain variations in work. Recently, however, Blelloch [2] described a technique known as *flattening* to reduce nested parallelism to vector operations. The technique is formalized as a set of program transformations in [13] and used in a number of high level languages.

The result of flattening (1.1) is

```
binsearch^1(distribute(A,length(S)),S)       (1.3)
```

where `distribute` function generates a copy of the source sequence `A` for each element of `S` and `length` returns the number of elements in a sequence. The function `binsearch`^{1} is a data-parallel version of binary search with the signature $Seq(Seq(Int)) \times Seq(Int) \rightarrow Seq(Int)$ that was constructed by flattening the user-defined `binsearch` function. Each step in `binsearch`^{1} corresponds to one simultaneous step of all invocations of `binsearch`, and is implemented with a constant number of vector operations.

The flattening technique can yield competitive code. Blelloch *et. al*. reported that "[Compiled NESL programs] perform[ed] competitively with native code for regular data and often superior on irregular data." [4]. The same portable NESL program was compared against optimized CM-Fortran code on a CM-2 and Fortran77 on the Cray C90 and in some cases performed better by an order of magnitude.

1.3 Work inefficiency

While the technique of flattening nested parallelism is sound [6, 14], it incurs some practical problems in its application. In (1.3), `A` is replicated to match the number of values in `S` in order to agree with the signature for `binsearch`^{1}. As a result `binsearch`^{1} is supplied a distinct copy of `A` for each value of `S`. We would expect that the total work in evaluating (1.1) would be $O(|S| \log_2 |A|)$, but the replication of `A` already requires $O(|S| \bullet |A|)$ work to be performed, so (1.3) is not work efficient. The problem is that each instance of `binsearch` is indexing a separate copy of `A`; this is not necessary if concurrent reads are permitted. The replication of indexed values is a general problem in the flattening of nested data-parallelism and not limited to our implementation. Blelloch reports this increased work complexity in NESL and recommends that the programmer be aware of the problem and not use indexing in sequence comprehensions [3, appendix C]. To aid the programmer in avoiding such uses of indexing, NESL provides many primitives that, in parallel, select values from sequences in common access patterns.

These support primitives maintain work efficiency, but because they must be applied *outside* of iterators, they interfere with function modularity. Flattening expression (1.1) yields work-inefficient code because the function `binsearch` contains an indexing operation. However, avoiding the inefficiency by using the selection primitives requires the programmer to move pieces of the function body across the function boundary and outside the surrounding iterator. Because the selection operations can depend on other function parameters and variables within the function's scope, all functions that contain indexing must be specialized *for each invocation of the function*

occurring inside an iterator. This is extremely cumbersome and difficult to implement, yet necessary in order to generate efficient code.

1.4 Contributions

Our goal is to generate work efficient code for all data parallel expressions. This paper addresses the problems that arise from the data-parallel application of indexing operations and functions that contain indexing. In section 2 we give an overview of the flattening process and present new flattening transformations that improve on those we presented in [13]. In section 3 we describe three approaches to eliminating or reducing the cost of replications in the generated code. The most important of these, work-efficient indexing, solves the increase in asymptotic work complexity and is presented in section 4. Finally, in section 5 we present our experimental results using these techniques.

2. Flattening nested data-parallelism

2.1 Overview

The data-parallel subset of Proteus that is translated to vector operations is applicative and includes a single assignment construct (`let` ... `in` ...), a value-returning conditional expression (`if` ... `then` ... `else` ...) and the iterator construct (`[x in D: e]`). The values in the subset are numbers, tuples, nested sequences and functions. Nested sequences are homogeneous and of fixed depth. The subset can be statically typed.

All parallelism in the Proteus data-parallel subset is specified using the iterator construct. Flattening is accomplished by eliminating iterators. This is done by defining rules to transform iterators through all the constructs of the Proteus language. Iterators surrounding the leaves of an abstract syntax tree(AST) are replaced by equivalent iterator-free sequence operations. In this fashion all iterators can be eliminated from a Proteus program, and the resulting program is expressed entirely in terms of a small number of basic data-parallel sequence operations (see figure 1). Most data-parallel operations are implemented using a constant number of vector operations, some require O(depth of sequence) vector operations.

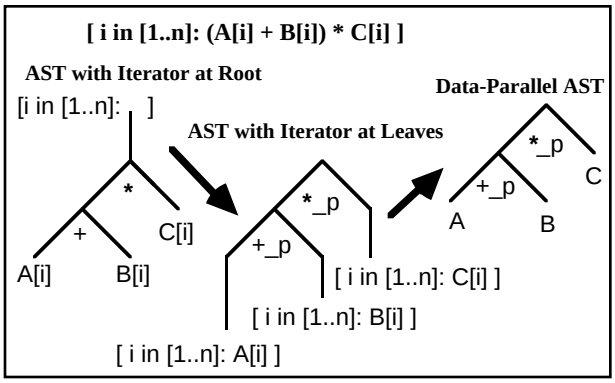

Figure 1

When an iterator is transformed through a function application, a data-parallel version of the function must be

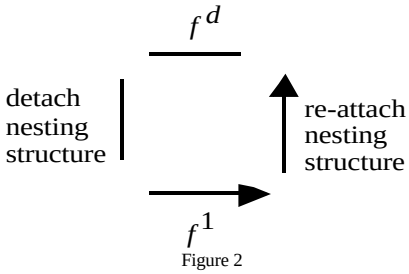

Figure 2

applied. Instead of repeatedly evaluating single argument values, the data-parallel function evaluates sequences of arguments in parallel. Parallel versions of all primitives have been implemented directly with vector operations, but data parallel versions of user-defined functions that appear in iterators must be generated through additional transformations. To do so, an iterator is placed around the function body and the same iterator-eliminating transformations are applied, removing the introduced iterator and yielding the data-parallel version of the function.

For an arbitrary function *f*, we use f^1 to denote the data-parallel version which applies *f* to all elements of a sequence in parallel. Signatures of the data-parallel versions are related to the original function definition by:

$$f^n : \alpha \rightarrow \beta \quad \text{implies} \quad f^{n+1} : Seq(\alpha) \rightarrow Seq(\beta) \qquad (2.1)$$

f^d indicates a version of *f* that applies *f* in parallel to all elements at the d^{th} level of a nested sequence; its signature is $Seq^d(\alpha) \rightarrow Seq^d(\beta)$. Fortunately, since the parallel function applications are independent, we can avoid the need for f^d where $d>1$ by peeling away the nesting structure of the arguments and using f^1 in all contexts. (see figure 2)

Data-Parallel Library Primitives		Signature of Basic Operation	Time	Work
`arith-ops`	arithmetic & logical operations	$Num \times Num \rightarrow Num$	O(1)	O(n)
`index`	extract an element from a sequence	$Seq^{d}(\alpha) \times (Int_1 \times ... \times Int_k) \rightarrow Seq^{d-1}(\alpha)$	O(d)	O(dn)
`length`	count the elements in a sequence	$Seq(\alpha) \rightarrow Int$	O(1)	O(1)
`distribute`	replicate values to form a sequence	$\alpha \times Int \rightarrow Seq(\alpha)$	O(1)	O(nr)
`range`	enumerate an integer interval	$Int \times Int \rightarrow Seq(Int)$	O(1)	O(n)
`restrict`	pack a sequence according to a mask	$Seq(Bool) \times Seq(\alpha) \rightarrow Seq(\alpha)$	O(d)	O(n)
`combine`	merge two sequences based on a mask	$Seq(Bool) \times Seq(\alpha) \times Seq(\alpha) \rightarrow Seq(\alpha)$	O(d)	O($n+m$)
`extract`	peel off a sequence's nesting structure	$Seq^{k}(\alpha) \times Int \rightarrow Seq^{k-n}(\alpha)$	O(1)	O(1)
`insert`	reattach nesting structure to a sequence	$Seq^{k}(\alpha) \times Seq^{d}(\alpha) \times Int \rightarrow Seq^{k+n}(\alpha)$	O(1)	O(1)

Table 1: n, m = number of elements in a sequence; d,k = depth of a sequence; r is the number of copies. Work and time complexity are given for the data-parallel versions of the primitives and are specified in the vector model where scan and reduce are O(1) time operations.

2.2 Nested sequence primitive operations

The data parallel operations remaining after application of the rules constitute the instruction set of an abstract data-parallel machine. We have implemented this abstract machine using C and a C-callable Data Parallel Library (DPL) [12] to support nested sequences and their operations. All nested sequence operations in the abstract machine specify work proportional to the length of the nested sequence and time either constant or proportional to the depth of the nested sequence (see table, note that the work and time complexity are specified in terms of the vector model where scan and reduce are O(1) time operations). DPL is implemented with vector operations provided by the C Vector Library (CVL) [5]. DPL provides both parallel execution and architecture independence by building upon these same features in CVL. Hence transformed programs may be run efficiently on a wide variety of machines, including the Cray C90, the TMC CM5, the MasPar MP-2 and UNIX workstations.

In general, the signature of the data-parallel versions of these operations can be derived using (2.1). For example, the signature of the data-parallel version of the `distribute` operation, `distribute`^{1}, is $Seq(\alpha) \times Seq(Int) \rightarrow Seq(Seq(\alpha))$.

We present improved transformation rules to flatten nested data-parallelism in Proteus. These new rules reflect an approach that transforms a single iterator at a time which differs from our previous strategy of transforming groups of iterators [13]. When iterators are nested, these rules are first applied at the innermost iterator, eliminating it but possibly introducing new function calls. The rules are then applied to the new innermost iterator, and repeated until no iterators remain.

These are the *baseline transformation rules* and we will use them for comparison against the new rules we introduce.

Rule 0: Eliminate restriction clauses in iterators

$[V \texttt{ in } D \mid c : e] \equiv [V \texttt{ in } \texttt{restrict}(c,D) : e]$

Rule 1: Distribution of iterator over function application

$[V \texttt{ in } D : f^{n}(e_1,...e_k)] \equiv$
$f^{n+1}([V \texttt{ in } D : e_1], \ldots, [V \texttt{ in } D : e_k])$

Rule 2: Distribution of iterator over single assignment

$[v \texttt{ in } D :$ **let** $t = e_1$ **in** $e_2]$ $\equiv$
let
$T = [v \texttt{ in } D : e_1]$
in
$[i \texttt{ in } [1..\#T] : e_2]$
with every occurrence of t in e_2 replaced by $T[i]$ and every occurrence of v in e_2 replaced by $D[i]$

Rule 3: Distribution of iterator over conditional statement

$[v \texttt{ in } D :$ **if** e_1 **then** e_2 **else** $e_3]$ $\equiv$
let
$M = [v \texttt{ in } D : e_1]$
$T = [k \texttt{ in } \texttt{restrict}(M,D) : e_2]$
with every occurrence of v in e_2 replaced by k.
$E = [k \texttt{ in } \texttt{restrict}(\texttt{not}(M),D) : e_3]$
with every occurrence of v in e_3 replaced by k.
in
`combine`(M,T,E)

Rule 4: Iterator Elimination

$[v \texttt{ in } D : v] \equiv D$
$[v \texttt{ in } D : u] \equiv$ `distribute`$(u,$ `length`$(D))$
where u is a constant or variable

Rule 5: Introduction of data-parallel functions

For each function:
function $g(x_1, \ldots, x_n) =$ `ret`
generate:

```
function g^1(V_1, ... , V_n) =
    return
        if empty(V_1) &&...&& empty(V_n)
        then []
        else [i in [1..#V_1]: e ]
```

Rule 6: Implementation of deep data-parallel functions

$$
\begin{aligned}
&f^d(e_1, e_2, \ldots, e_n) \equiv \\
&\quad \textbf{let} \quad V_1 = e_1, \ldots, V_n = e_n \\
&\quad \textbf{in} \\
&\qquad \texttt{insert} \\
&\qquad\quad (f^1 \quad (\quad \texttt{extract}(V_1, d-1), \\
&\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\quad \texttt{extract}(V_n, d-1)), \\
&\qquad\quad V_1, d\text{-}1)
\end{aligned}
$$

3 Reducing replication costs

3.1 Expression hoisting

Iterator expressions that are either completely or partially independent of the iterator variable are replicated unnecessarily causing inefficient evaluation and in many cases, increased asymptotic work complexity. We use expression hoisting, a well-established technique from code optimization, to move expressions outside of the iterators that they do not depend upon. A new transformation rule allows the expression to be evaluated once and only its result is replicated.

$$
\begin{aligned}
&[v \ \textbf{in}\ D\texttt{:}\ e\ ] \equiv \quad \textbf{let} \quad x = e \\
&\qquad\qquad\qquad\quad \textbf{in} \quad [v \ \textbf{in}\ D\texttt{:}\ x]
\end{aligned}
$$

where e is a non-simple expression independent of v

Consider the following example.

```
R = [ v in D: [ w in E: 3 + v]]                (3.1)
```

Since the innermost expression is independent of the inner iterator, it can be hoisted out.

```
R = [ v in D:    let    x = 3 + v
                 in  [ w in E: x]]
```

The code generated by transforming (3.1) without code hoisting,

```
n = length(D);
T = length^1(distribute(E,n));
R = plus^2(distribute^1(distribute(3,n),T),
                              distribute^1(D,T));
```

specifies two `distribute`^{1} operations, which are implemented with costly general communication. By transforming (3.1) using the code hoisting transformation, we generate:

```
T = plus^1(distribute(3, length(D)), D);
R = distribute^1(T, length^1(distribute(E,
                                  length(T))));
```

This requires only one general communication and specifies less work because the addition is performed at a lower multiplicity and its result is replicated.

3.2 Efficient scalar replication

scalar V_k replicated by $V_k[i]$

Consider a nesting of iterators around a constant.

```
[ i in [1..n]: [ j in [1..i]:c ]]   (3.2)
```

The transformation rules transform (3.2) to the following.

```
distribute^1(distribute(c,length(range(1,n))),
                          length1(range(1,n)));
```

This specifies an inefficient, multi-stage, replication that requires general communication. Since the value being replicated is a scalar, regardless of the shapes of the intermediate sequences and the number of replications, we can implement the operation with a single, inexpensive broadcast (see figure 3).

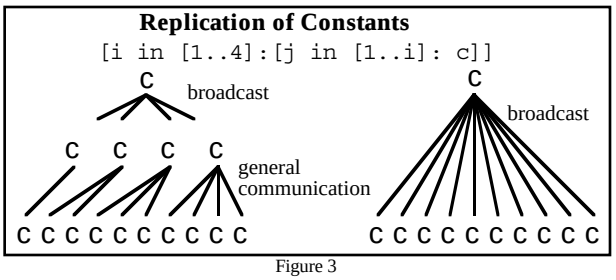

Figure 3

We introduce the `promote` operation, which has a signature of $Num \times Seq^k(Int) \rightarrow Seq^k(Num)$. It replicates a scalar input using broadcast communication to the same size and nesting structure as its second parameter. For example `promote(3,[[1,2,9],[5,8]])` yields `[[3,3,3],[3,3]]`.

To support this operation, we must introduce two new transformation rules: the first specializes baseline transformation rule 4 for scalars and the second specializes rule 1 for the `promote` function.

```
[v in D: c ] ≡ promote(c,D)
[v in E: promote(c,D) ] ≡
                  promote(c,[v in E:D])
```

These new rules transform expression (3.1) to simpler, more efficient generated code

```
promote(c,range^1(distribute(1,n),range(1,n)))
```

which avoids general communication.

3.3 Work efficient indexing operation

An index operation selects a value from a nested *source* sequence based on a set of indices. If we represent a nested sequence as a tree, as in figure 4, then the height of the tree corresponds to the depth of the source sequence and each index is a tuple that specifies a path through the

tree. From a given node, the next index specifies to which child node the index path continues. The result of an indexing operation is a nested sequence represented by the sub-tree rooted at the node where the path terminates.

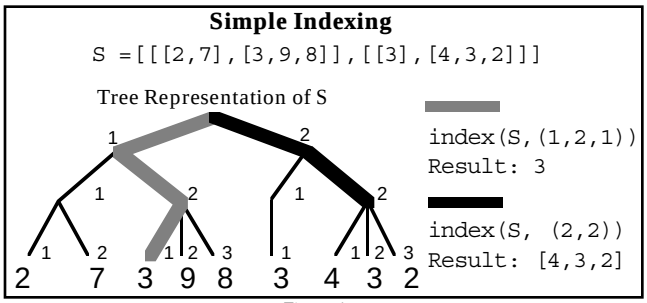

Figure 4

To perform multiple indexing operations in parallel, at each step we advance along the next edge of all index paths through the tree simultaneously. We repeat this operation until we have exhausted the indices, yielding a sequence of results.

The straightforward implementation of indexing has the signature $Seq(Seq(\alpha)) \times Seq(Int \times ... \times Int) \rightarrow Seq(\alpha)$. We call this `rep_index`, because the source sequence is replicated. Using it with the transformation of the following expression, `[v in D: A[v] ]`, yields

```
rep_index^1(distribute(A, length(D)),(D))   (3.3)
```

This operation creates a copy of the sequence `A` to positionally correspond with each value in `D`. This correlation specifies that the i^{th} index of `D` is used to select an element from the i^{th} copy of `A`. In cases where multiple iterators surround an indexing operation, the source sequence will be replicated by the *product* of the sizes of the iterator domains. The positional correspondence is the source of the increased asymptotic work complexity and simply too costly to maintain.

We avoid the work inefficiency by defining a new indexing operation that removes the correspondence requirement. We call it `eff_index` and it has the signature $Seq(\alpha) \times Seq(Int \times ... \times Int) \rightarrow Seq(\alpha)$. Instead of replicating the source sequence, it is shared among all the elements of the index sequence. We can express `[v in D: A[v]]` without the increased work complexity of (3.3), as

```
eff_index^1(A, (D))
```

In practice, source sequences may be dependent on some iterators and independent of others. When a source is dependent on a surrounding iterator, such as the transformation of expression (1.2), the source sequence must be evaluated according to the dependency, increasing its depth and generating many different sub-sequences. In this case, there is an implicit positional correlation between the source sequence and the sequence of indices that must be retained.

To handle this we observe that `eff_index` can emulate positional correspondence by introducing another index. The new index is an enumeration to the size of the domain of the iterator. For example if `B` is `[[1,2,3],[4,5,6,7],[8,9],[0]]`, then the expression `[i in [1..4]: B[i][1] ]`, which yields `[1,4,8,0]`, can be expressed using either implementation of index.

```
rep_index^1(distribute(B,4), ([1,1,1,1]))

eff_index^1(B, ([1,2,3,4], [1,1,1,1]))      (3.4)
```

Thus we add another transformation rule 1, that treats the function index as a special case.

$[v$ **in** D: `eff_index`$^d(E, (e_1,...,e_k))$ $]$ $\equiv$
if E depends on v
`eff_index`$^{d+1}$`([`v **in** D: E`],(extend(`D`,`e_1`),`
`[`v **in** D: e_1`],..,[`v **in** D: e_k`]))`
otherwise
`eff_index`$^{d+1}$`(`E`, ([`v **in** D: e_1`],..,`
`[`v **in** D: e_k`]))`

We again have introduced a primitive to support the new rule. The `extend` operation has a signature of $Seq(Int) \times Seq^k(Int) \rightarrow Seq^{k+1}(Int)$. It extends each element of the first sequence to the size and shape of the second sequence. We use it to insure that introduced indices, such as the enumeration in expression (3.4), will conform to existing indices. Note that `extend` is transformed as any arbitrary function.

4 Implementation of efficient Indexing

4.1 Colliding indices

Parallel indexing using `rep_index` performs multiple simple indexing operations by following index paths through separate copies of the source index tree. We have introduced new transformations that avoid replicating the source sequence. As a result, performing parallel indexing with `eff_index` follows all index paths through the same source tree guaranteeing that the indices will collide. On the first step of a parallel index operation, there are as many attempts to access the root of the tree as there are indices. These colliding accesses may lead to inefficient parallel execution because of the implicit costs of concurrent reads.

4.2 Node extension

To make work efficient indexing viable, we must address the concurrent reads. We present a workable compromise using a technique called *node-extension*. Instead of replicating the full representation of a nested sequence, nodes within the tree representations are replicated different amounts. After the replication, each

level in the tree will have O(# of indices) nodes. If, at some level in the tree, there are *n* nodes, each of these nodes is replicated by (1/*n* * # of indices). When this product is less than or equal to 1, no replication is done. The resulting structure is a *node-extended tree* (see figure 5) that reduces the number of concurrent reads, but maintains the asymptotic work complexity because the replication is bounded by the number of the indices.

Indexing using a node-extended tree representation for nested sequence requires that the index be adjusted by a random value. When extending the index path, there may be many edges that lead to the "same" child. By randomly distributing the access requests for a specific node among its copies, we reduce the number of concurrent reads. With this structure, on the first step of parallel indexing there are still O(# of indices) attempts to access the root, but now there are an equal number of copies of the root available and it is expected that the number of concurrent reads will be small. Clearly the performance is dependent on the characteristics of the indices, but, as our experimental observations bear out, the additional copies and the randomizing factor yield better performance.

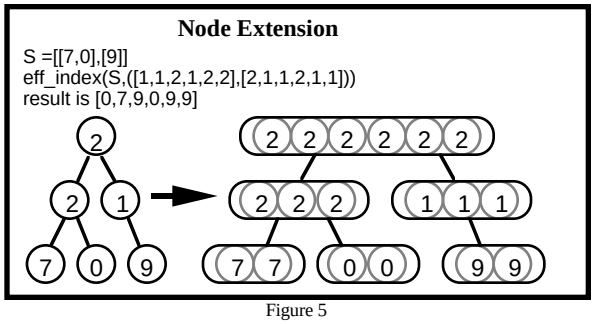

Figure 5

4.4 Interfunction indexing

To achieve work efficient performance, source sequences for index operations must not be replicated by any independent iterator, even those outside the invocation of a function that contains indexing operations. The baseline transformation rule for function application replicates all the parameters, sabotaging any subsequent indexing transformation. When a function parameter is used only as the source sequence to an indexing operation within the function, this information must be propagated up to the level of the function definition. Then, a new transformation rule that does not replicate source sequence parameters, can be applied.

```
[v in D: f^d(only_indexed(A))]  ≡
    if A depends on v
        f^(d+1)(only_indexed([v in D: A],D))
    otherwise
        f^(d+1)(only_indexed(A))
```

```
[v in D: f^d(only_indexed(A,e))]  ≡

    if A depends on v
        f^(d+1)(only_indexed([v in D: A],
                extend(D,e),[v in D: e]))
    otherwise
        f^(d+1)(only_indexed(A,[v in D: e]))
```

A new primitive, `only_indexed`, accumulates any introduced indices due to dependency on iterators, as in 3.4, and attaches them to the representation of the source sequences. This allows the necessary information to traverse a function boundary without violating the function's modularity. Inside the function, the indexing operation retrieves the added indices and applies them to the source sequence prior to any explicitly expressed indices.

Whether to wrap an invocation of `only_indexed` around an actual parameter of a function call is determined through static analysis of the function definition. If a parameter is only used as source sequence of indexing either in body of the function being examined or in the body of some subsequently invoked function, it is marked as only indexed.

These transformations provides work efficient execution for functions that directly call indexing, but they are also applicable to functions that indirectly call indexing through other functions. The source dependency can be propagated up to the top level of the program and all replication of the sequence can be avoided. This transformation is needed to avoid replication in the calls to `binsearch`.

5. Discussion

5.1 Results

We generated parallel code for the parallel binary search expression (1.1),

```
[x in S: binsearch(A,x)]
```

two different ways: first using the baseline transformations and replicating indexing, and then using the additional transformations presented in this paper and work-efficient indexing with node-extension.

The first approach generated code with asymptotic work complexity $O(|S| \cdot |A| \log_2 |A|)$, while the second strategy produced code with asymptotic work complexity $O(|S| \log_2 |A|)$ with some additional costs due to unavoided concurrent reads. The source sequence `A` consisted of an enumeration of positive integers up to the specified size. The search values in `S` consisted of random values in the range of the source sequence. The algorithm was written recursively using Proteus and then transformed to C and nested sequence operations implemented using DPL.

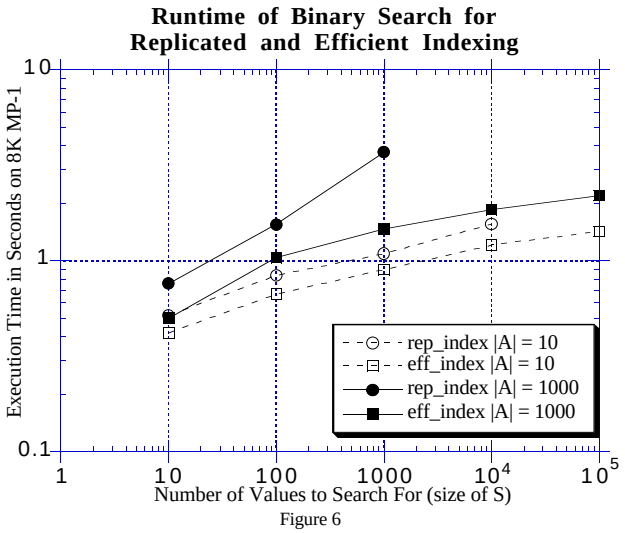

Figure 6

Figure 6 gives the running times of the resultant codes for several different sizes of A and S on an 8K processor MasPar MP-1.

In figure 6 the performance is compared at two values of |A|. When |A| = 10, since |S|>>|A|, the number of concurrent reads would be high without the node extension strategy. Even in this setting, the work-efficient indexing performed better at all values for |S| than the replicated indexing. This indicates that the node replication strategy is highly effective in keeping the number of concurrent references small.

For fixed |A|, the performance is linear or even sublinear in |S|. By avoiding the replication of A, the node-extension version was also able to execute much larger problem sizes. This is clearly visible as |A| is increased to 1000 and |S| two orders of magnitude larger than the maximum size rep_index can easily handle. At larger values of |A| the additional work due to replication becomes a dominant factor in the cost.

5.2 Conclusions

Nested sequences and fully general nested apply-to-all constructs dramatically simplify the expression of complex parallel computations. The flattening technique enables such expression to be translated to efficient and highly parallel vectors operations.

Since one of the key features of this style of parallel programming is that it permits the succinct expression of irregular but work-efficient parallel computations, it is particularly important that all constructs of the notation have the optimal work efficiency. Indexing is a particularly important and practical operation for which this property had not yet been achieved.

Our technique provides the opportunity to achieve the improved performance without sacrificing the expressive and conceptually intuitive indexing operation. We have described an approach to realize work-efficient data-parallel indexing based on bounding the amount of replication and randomly dispersing concurrent reads. We are currently applying these techniques to large algorithms and are highly encouraged by our preliminary results.

Bibliography

[1] J. Adams, W. Brainerd, J. Martin, B. Smith and J. Wagener, Fortran 90 Handbook, Intertext-McGraw Hill, 1992.

[2] G. Blelloch, Vector Models for Data-Parallel Computing, The MIT Press, 1990.

[3] G. Blelloch, "NESL: A Nested Data-Parallel Language(2.6)," Technical Report CMU-CS-93-129, Carnegie Mellon University, 1993.

[4] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Proceedings of Fourth ACM Symposium on Principles and Practices of Parallel Programming*, May 1993.

[5] G. Blelloch, S. Chatterjee, J. Sipelstein and M. Zahga, "CVL: A C vector library," Draft Technical Report, Carnegie Mellon University, March 1993.

[6] G. Blelloch and G. Sabot, "Compiling Collection-Oriented Languages onto Massively Parallel Computers," *Journal of Parallel and Distributed Computing*, 1990.

[7] R. Faith, L. Nyland, D. Palmer, and J. Prins, "The Proteus NS Grammer," Technical Report TR94-029, UNC-CH, 1994.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, Fortran D Language Specification, Report COMP TR90-141(Rice) and SCCS-42c (Syracuse), Rice University and Syracuse University, 1991.

[9] A. Goldberg, J. Prins, J. Reif, R. Faith, Z. Li, P. Mills, L. Nyland, D. Palmer, J. Riely, and S. Westfold, *The Proteus System for the Development of Parallel Applications*, in M. Harrison, editor, Prototyping Technologies: The ARPA ProtoTech Project. Kluwer Academic Publishers, to appear.

[10] High Performance Fortran Forum, "High Performance Fortran Language Specification," January, 1993.

[11] T. More, "The Nested Rectangular Array as a Model of Data," *APL79 Conference Proceedings.* ACM 1979.

[12] D. Palmer, "DPL - Data Parallel Library Manual," Technical Report TR93-064, UNC-CH, November, 1993.

[13] J. Prins and D. Palmer, "Transforming High-Level Data-Parallel Programs into Vector Operations," *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and*

Practice of Parallel Programming, San Diego, CA, p. 119-128, 1993.

[14] J. Riely, S. Purushothoman Iyer, and J. Prins, "Compilation of Nested Parallel Programs: Soundness and Efficiency," Technical Report, UNC-CH, 1994.

[15] G. Sabot, The Paralation Model : Architecture-Independent Parallel Programming, MIT Press, 1988.

[16] J. Schwartz, "Set Theory as a Language for Program Specification and Programming," Technical Report Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1970.