DPL : Data Parallel Library Manual

Technical Report: UNC-TR93-064
Original Version: November 2, 1993
Revised: March 4, 1994

Daniel W. Palmer
Department of Computer Science, University of North Carolina
Chapel Hill NC 27599-3175 (919)-962-1968
palmerd@cs.unc.edu

0. Introduction

In [PP93] we described a transformational approach to realizing architecture independent parallel execution of a high-level parallel language. Detailed in that document is a series of steps that when applied to high-level, data-parallel programs written in the Proteus programming language yield parallel execution on a variety of different parallel architectures. The Data Parallel Library (DPL) directly supports Proteus by supplying a vital link in the transformational execution system.

1. DPL - Description and Requirements

The Data Parallel Library is a collection of C-callable routines that provide the capability to treat nested sequences as a primitive type. DPL supports data-parallelism by allowing each operation to be applied in parallel to a sequence of inputs yielding a sequence of results. The library contains a wide variety of functions, from the simple display of nested sequence characteristics to complex multi-level indexing of nested sequences (the complete list of operations is detailed in section 4 and appendix A).

The library is designed specifically to be used with C as an executable target notation for transformations of Proteus programs. To meet this goal the library must satisfy three criteria: it must be portable across many parallel architectures, it must perform nested sequence operations in parallel and it must present a simple and accessible calling interface.

Portability

The library must be portable so that Proteus can achieve architecture independence. A significant investment in the creation of a parallel program should not be negated by a change or upgrade in hardware. DPL is portable because it is fully implemented in terms of vector operations, which are well-suited for parallel execution on many different architectures. The vector model implementation that DPL uses is called the C Vector Library (CVL). It is fully detailed in [BCS+90] and pertinent characteristics are briefly described in section 2. By extension, DPL can operate on any architecture that CVL has been implemented on.

Parallelism

The library must execute in parallel so that the parallel algorithms that are expressed in Proteus can attain actual parallel execution. We wish to abstract away the details of parallel hardware, not the performance. DPL supplies fully parallel operations by implementing all nested sequence operations in terms of the vector model which is supported by many parallel architectures. By implementing operations on nested

sequences in terms of vector operations, DPL can perform these operations in O(sequence-depth) time, independent of sequence length.

Accessibility

The library routines must be transparent in the sense that the transformation process should not be concerned with low-level issues of execution. Detailed mechanisms of memory management, representation of nested sequences and optimization of operations on nested sequences should be assumed by the transformation process. DPL hides the irregularities and idiosyncrasies of the underlying software and hardware from higher levels in the execution hierarchy.

2. Properties of CVL

CVL is a set of C callable routines that present an abstract vector model of a parallel machine. The vector model is a description of an abstract processor that operates on vectors of arbitrary length as primitive entities. It is fully described and explored in [Blel90].

Portability

CVL was written to be the back-end for the data-parallel language NESL [Blel92]. As such, it was designed to port to different architectures with relative ease. Currently there are CVL implementations for the Cray-YMP, Thinking Machine's CM-2 and CM-5, MasPar's MP-1[FHS93] and a serial version for workstations. By simply implementing all of DPL in terms of C and CVL satisfies DPL's portability requirement. DPL will run on all machines that currently support CVL and, as CVL is ported to new architectures, DPL will be automatically ported to those by extension.

Parallelism

All of the vector operations provided by the versions of CVL developed for parallel machines are executed in parallel. CVL makes use of the native parallel operations available on different architectures to provide real parallel work and time measurements for algorithms developed in NESL. Through judicious use of CVL operations and suitable structures for the representation of nested sequences, DPL provides, by extension, parallel work and time measurements for algorithms developed in Proteus.

Accessibility

As repeatedly stated in the CVL manual, the library is purposefully low-level; the programmer is responsible for the context in which the library operations are invoked. Many facets of the library that make it easy to port, make it burdensome to program. The library provides no error checking, no vector memory management and no necessary scratch space for vector operations. These features must be supplied by the CVL user. DPL insulates the programmer from these low-level concerns and provides a higher level interface for parallel execution.

3. DPL Operations

There are four classes of DPL operations supplied by the library: elementwise, representation manipulation, sequence and auxiliary. This section describes each of these classes in turn.

Elementwise operations

Elementwise operations perform basic arithmetic and logical operations on a single flat sequence or between two flat sequences. By flat sequence, we mean a sequence of non-aggregates -- base values. These

operations are simply extensions of ordinary arithmetic and logical operations, applied to a collection of values. Many of these DPL operations are directly implemented with corresponding CVL operations. The table below gives the characteristics of each elementwise operation. The entries consists of the name of the operation, the number of input sequences the operation requires, the types of both the inputs and the outputs and a single example of the operation.

Function	#	Input type(s)	Output type	Inputs	Result
add_p	2	numeric	same as input	[1,2],[4,6]	[5,8]
sub_p	2	numeric	same as input	[7,3],[2,6]	[5,-3]
equal_p	2	any	boolean	[1,7],[4,7]	[F,T]
min_p	2	numeric	same as input	[4,9],[6,5]	[4,5]
max_p	2	numeric	same as input	[4,9],[6,5]	[6,9]
div_p	2	numeric	real	[1,9],[2,3]	[0.5,3.0]
mul_p	2	numeric	same as input	[2,4],[3,2]	[6,8]
mod_p	2	integer	integer	[3,9],[2,5]	[1,4]
grt_p	2	numeric	boolean	[9,3],[8,3]	[T,F]
less_p	2	numeric	boolean	[3,7],[5,6]	[T,F]
grteq_p	2	numeric	boolean	[7,1],[5,1]	[T,T]
nequal_p	2	any	boolean	[4,5],[8,5]	[T,F]
not_p	1	boolean	boolean	[F,T,T,F]	[T,F,F,T]
abs_p	1	numeric	same as input	[-3,4]	[3,4]
or_p	2	boolean	boolean	[T,F],[F,F]	[T,F]
and_p	2	boolean	boolean	[T,F],[T,T]	[T,F]
sqrt_p	2	numeric	real	[4.41,25]	[2.1,5.0]
sin_p	1	real	real	[0, π/2]	[0.0,1.0]
cos_p	1	real	real	[0, π/2]	[1.0,0.0]
random_p	1	numeric	same as input	[7,7,43,12]	[4,2,34,3]
dbl_p	1	integer	real	[4,2]	[4.0,2.0]
int_p	1	boolean	integer	[T,F]	[1,0]
boo_p	1	integer	boolean	[1,0]	[T,F]

Representation manipulations

Manipulations of the representation are critical in the implementation of nested data-parallelism. They allow the application of a function designed to operate on flat sequences to be applied to nested sequences of any depth. This is the key step in the transformational execution of Proteus programs. For example, two flat sequences of integers can be added together in parallel by simply using the `add_p` operation described in the previous section. Suppose instead, that we wish to add all the corresponding elements of two depth-two sequences. We can write a function that explicitly handles this case. However, that solution would not be general and requires an arbitrary number of different routines to handle all possible sequence depths in a given program. Instead, we observe that the elementwise addition of corresponding values in a depth two sequence can be accomplished by momentarily ignoring the nesting information and treating the nested sequence as if it were flat. This is precisely what the representation manipulations do. The operation,

`extract` takes a nested sequence as input and removes a specified number of nesting levels. The inverse operation, `insert`, takes two nested sequences and puts a portion of the nesting context of one sequence "around" the other. By coordinating these two operations, all or part of the nesting structure of a nested sequence can be temporarily ignored, while an elementwise operation is applied to its values.

EXTRACT

Description: Removes a specified portion of the nesting structure, yielding a sequence with the same data values, in a shallower nesting configuration.

Function versions with signatures:

`extract` $\quad \text{Seq}^{k}(\alpha) \times \text{Int} \rightarrow \text{Seq}^{k-d}(\alpha)$ `where 0 < k-d < k`

Example applications:

```
extract([[0,2,3],[5,7]],1)
        yields: [0,2,3,5,7]
extract([[[2,3],[0],[1,7,5,9,8]],[[6,34,-4]]],1)
        yields: [[2,3],[0],[1,7,5,9,8],[6,34,-4]]
extract([[[2,3],[0],[1,7,5,9,8]],[[6,34,-4]]],2)
        yields: [2,3,0,1,7,5,9,8,6,34,-4]
```

INSERT

Description: Places the first input sequence within a specified portion of the nesting structure of the second input sequence.

Function versions with signatures:

`insert` $\quad \text{Seq}^{k}(\alpha) \times \text{Seq}^{n}(\alpha) \times \text{Int} \rightarrow \text{Seq}^{k+n-d}(\alpha)$

Example applications:

```
insert([1,2,3,4,5],[[8,7,5],[2,3]],1)
        yields: [[1,2,3],[4,5]]
```

EXTRACT and INSERT used in combination

Description: Temporarily ignores the nesting structure so an elementwise operation can be applied to the nested sequence values.

Example applications:

```
insert(add_p(extract([[4,2,3],[5]],1),extract([[5,3,1],[6]],1)), [[0,2,3],[5]],1)
        yields: [[9,5,4],[11]]
insert(abs_p(extract([[[-2,3]],[[-2],[-4,5]]],2)), [[[-2,3]],[[-2],[-4,5]]],2)
        yields: [[[2,3]],[[2],[4,5]]]
```

FLATTEN

Description: Shorthand for calling extract with an extraction depth of 1.

Function versions with signatures:

```
flatten_s      Seq^k(α) → Seq(α)
flatten_p      Seq(Seq^k(α)) → Seq(Seq(α))
```

Example applications:

```
flatten_s([[4,2,3],[5]])
       yields: [4,2,3,5]
flatten_p([[[2,3],[7,1,8]],[[-2],[-4,5]]])
       yields: [[2,3,7,1,8],[-2,-4,5]]
```

DEEPEN

Description: Makes each sub-sequence of a nested sequence one level deeper.

Function versions with signatures:

```
deepen_s       Seq^k(α) → Seq^(k+1)(α)
deepen_p       Seq(Seq^k(α)) → Seq(Seq^(k+1)(α))
```

Example applications:

```
deepen_s([[4,2,3],[5]])
       yields: [[[4,2,3],[5]]]
deepen_p([[2,3],[7,1,8],[-2],[-4,5]])
       yields: [[[2,3]],[[7,1,8]],[[-2]],[[-4,5]]]
```

MAKE REPRESENTATION

Description: Shorthand for calling extract with an extraction depth of 1.

Function versions with signatures:

```
mke_rep_i      Int× Seq^k(Int) → Seq^k(Int)
mke_rep_r      Real× Seq^k(Int) → Seq^k(Real)
mke_rep_b      Bool× Seq^k(Int) → Seq^k(Bool)
```

Example applications:

```
mke_rep_i(3,[1,2,3,4,5])
       yields: [3,3,3,3,3]
mke_rep_r(1.4,[[2,3],[7,4,2],[6,3,2,1]])
       yields: [[1.4, 1.4],[1.4, 1.4, 1.4],[1.4, 1.4, 1.4, 1.4]]
```

Sequence operations

Every DPL sequence operation has at least two forms, a parallel version and at least one non-parallel (or single) version. All operations have a parallel version, which is denoted with a "_p" suffix attached to the function root name. For each operation there are possibly many single versions, which can be denoted with the suffixes "_i" (for base-type, integer), "_r" (for base-type, real), "_b" (for base-type boolean), and "_s" (for sequence type). The valid single versions for each DPL operation are indicated in the complete list in this section.

To better illustrate the distinctions, consider the operations `distribute` and `range`. The `distribute` operation replicates an input value a specified number of times, creating a nested sequence with repeated values. Depending on what the type of the value being replicated is, a different version of the function is needed. To generate a sequence containing five 3's (`[3,3,3,3,3]`), the `dist_i` routine must be used, because the replicated input type is integer. If a sequence of 4 true values (`[T,T,T,T]`) is needed, `dist_b` must be used. Similarly with replication of sequences, `dist_s([1,2],3)` will generate the sequence: `[[1,2],[1,2],[1,2]]`. All four single version of the distribute routine are valid. This is not the case with the range operation. The `range_i` operation takes two integers and returns a sequence of integers beginning with the first and ending with the second. Since there is no canonical ordering of sequences, no discretized step value for real numbers and no more than two boolean values, there are no sequence, real or boolean single versions of the range operation. The following table contains the names of the basic sequence operations and which versions of the operations are valid. An 'i' in the versions column means that there is an integer version of the operation, etc. Also included is the signature for the parallel operation. Signatures for other versions can, in many cases, be mechanically derived from the parallel version.

Function	Versions	Signature of parallel version
base	sp	$Seq(Seq(\alpha)) \times Seq(Int) \rightarrow Seq(Seq(\alpha))$
build	irbsp	$Int \times Seq(\alpha)^{+} \rightarrow Seq(Seq(\alpha))$
combine	sp	$Seq(Seq(Bool)) \times Seq(Seq(\alpha)) \times Seq(Seq(\alpha)) \rightarrow Seq(Seq(\alpha))$
count_trues	sp	$Seq(Seq(Bool)) \rightarrow Seq(Int)$
cr_empty	sp	$Int \times Int \rightarrow Seq(Seq(\alpha))$
dist	irbsp	$Seq(\alpha) \times Seq(Int) \rightarrow Seq(Seq(\alpha))$
empty	sp	$Seq(Seq(\alpha)) \rightarrow Seq(Bool)$
index	irbsp	$Seq(Seq^{k}(\alpha)) \times Seq(Int)_1 \times \ldots \times Seq(Int)_d \rightarrow Seq(Seq^{k-d}(\alpha))$
length	sp	$Seq(Seq(\alpha)) \rightarrow Seq(Int)$
partition	sp	$Seq(Seq(\alpha)) \rightarrow Seq(Seq(Seq(\alpha)))$
range	i p	$Seq(Int) \times Seq(Int) \rightarrow Seq(Seq(Int))$
range1	i p	$Seq(Int) \rightarrow Seq(Seq(Int))$
reduce	irbsp	$(Func(\alpha \times \alpha) \rightarrow \alpha) \times Seq(Seq(\alpha)) \rightarrow Seq(\alpha)$
restrict	sp	$Seq(Seq(Bool)) \times Seq(Seq(\alpha) \rightarrow Seq(Seq(\alpha))$
split	sp	$Seq(Seq(Bool)) \times Seq(Seq(\alpha)) \rightarrow Seq(Seq(Seq(\alpha)))$
st_range	ir p	$Seq(Num) \times Seq(Num) \times Seq(Num) \rightarrow Seq(Seq(Num))$

In appendix A there is a detailed list of DPL sequence operations, including brief descriptions of their functions and a few examples. In these descriptions, the α symbol represents *any* legal type. This includes the basic types of integer, boolean and floating point, as well nested sequences of any depth. All of the examples clearly show what happens to a flat sequence under the given operation. In some cases there is a supporting example illustrating the effects of the operation on a nested sequence. There are no examples for nested sequences greater than depth two. This by no means implies that these operations are not supported. For any operation in which the type signature includes an α the operation can be applied to any type. This does not mean that there are versions of each operation for any depth sequence, but rather that a sequence of any depth can be viewed as a flat sequence of deeply nested elements. For example, the `index_s` operation returns a specified sub-sequence from a depth 2 sequence. If a sequence of depth 5 is supplied as

an input to the routine, it will be viewed as a depth 2 sequence of depth 3 sequences. The corresponding depth 3 sequence will be returned as a result. In other terminology, a Seq^{k}(Int) can also be viewed as $Seq^{2}(Seq^{k-2}(Int))$. Both views are consistent with the notation: Seq(α).

Auxiliary operations

The following operations are necessary for the proper execution of DPL, but do not fall into a easily labeled grouping. In every case, they are operations that facilitate the execution of other DPL operations, hence the name auxiliary.

FREE NESTED SEQUENCE

Description: Informs DPL that a particular nested sequence is no longer needed by the calling program and that the space it used is no longer required. DPL will then either reclaim the space if there are no other pointers to it or decrement the reference count if there are.

Calling form:

```
fre_nseq(nseq)
```

MAKE REDUCTION FUNCTION

Description: Takes single and parallel versions of an associative operation and creates a data structure that can be used in conjunction with the reduce operation to perform general reductions on sequences.

Calling form:

```
mke_red(func_i,func_p)
```

READ A NESTED SEQUENCE

Description: Reads from either a file or standard input a nested sequence or tuple and converts it to the representations that used in the library.

Calling form:

```
read_p(nseq)
```

SET UP MEMORY

Description: Allocates a single large block of vector memory from which to allocate space for nested sequences.

Calling form:

```
setmem()
```

MARK TEMPORARY VARIABLE STACK

Description: Sets an indicator that the anonymous reclamation should not go back further than a given point in the anonymous variable stack.

Calling form:

```
mark_temp_stack()
```

REMOVE MARK FROM TEMPORARY VARIABLE STACK

Description: Removes the previous indicator set with the mark_temp_stack() operation from the anonymous variable stack.

Calling form:

```
clear_temp()
```

REMOVE MARK FROM TEMPORARY VARIABLE STACK AND STORE RESULT

Description: Removes the previous indicator set with the mark_temp_stack() operation from the anonymous variable stack and moves supplied value off the stack..

Calling form:

```
keep_temp(nseq)
```

SHOW A NESTED SEQUENCE

Description: Generates a human readable format for nested sequences and displays it to a file or standard output.

Calling form:

```
show_p(nseq)
```

4. Memory Management

Designing the memory management portion of the Data Parallel Library has been a balancing act between generating a clear and usable interface and the "assembly language level" usage of the library. DPL is the back-end of the Proteus execution system and as such, must both provide a reasonably high-level interface and perform many low-level operations that are expressed at the language level. These conflicting requirements have lead me to provide two modes of memory reclamation for DPL: explicit and anonymous. Neither is ideal, both have their advantages. The first prohibits the generation of unreclamable nested sequences and the second provides additional routines for the automatic collection of them. Note that the two methods are semantically identical and only differ in where the responsibility for portions of the memory reclamation resides.

The user must inform DPL when an allocated nested sequence is no longer necessary and can be reclaimed. This is done by using the `fre_nseq` routine. Any unused memory that is not reclaimed, will accumulate and can prevent otherwise sound programs from running. The functional nature of DPL routines conflicts with this by allowing generation of anonymous nested sequences which *cannot* be explicitly reclaimed. For example:

```
a = dist_s(range_i(1,3),5);
...
fre_nseq(a);
```

The temporary nested sequence, `[1,2,3]`, created by the `range_i` operation does not have a handle, so it cannot be reclaimed using `fre_nseq`. Using explicit reclamation, the user can provide more effective memory use -- memory can be reclaimed exactly when it becomes unneeded -- but he must explicitly manage all temporaries. This mode simply disallows functional usage of DPL operations, taking advantage of the observation that any program written using anonymous function calls can be re-written with temporary variables and explicit freeing. To achieve the same semantics as the program segment in the previous example, we use:

```
temp = range_i(1,3);
a = dist_s(temp,5);
fre_nseq(temp);
...
fre_nseq(a);
```

Explicit reclamation provides the programmer with greater control and greater responsibility of memory usage.

Anonymous reclamation hides the temporary management from the user, allowing for much more compact code. However, the anonymous variables are no longer limited to the minimum possible life span and unreclaimed memory will accumulate between collections. Additionally, the user must specify to DPL which nested sequences are anonymous and which are not. Anonymous reclamation mode allows segments of the form shown in the original example, but provides an automatic method for reclaiming the anonymous variables. Additional routines are required to guide the automatic reclamation process. Each nested sequence that is generated must be identified as either anonymous or assigned. All generated nested sequences are assumed to be anonymous, unless explicitly indicated otherwise through use of the DPL `assign` operation. Anonymous nested sequences are automatically reclaimed, assigned nested sequences must be explicitly reclaimed by the programmer. The semantics of the `assign` operation are simply to return its single input parameter. Through side-effects it performs the reclamation of anonymous temporaries and changes its parameter from an anonymous variable to an assigned nested sequence. This is how the original example is expressed in anonymous mode:

```
a = assign(dist_s(range_i(1,3),5));
...
fre_nseq(a);
```

The `assign` operation indicates that the result of the `dist_s` is not anonymous and will be not be automatically reclaimed. It is associated with the variable `a`, and must be explicitly reclaimed. The result of the `range_i` operation is anonymous because no `assign` operation indicates otherwise, and will be automatically reclaimed. Note that the `assign` operation works the same for long chains of functional expressions, not just the simple case shown in the example.

The anonymous reclamation is implemented by making a list of all nested sequences allocated, removing those from the list that are assigned and then freeing all those that remain on the list. Effectively, an `assign` operation frees all temporaries generated since the previous `assign` operation. This straightforward scheme can fail when consecutive `assign` operations are separated by a function call. To prevent the reclamation of a temporary before its time, the anonymous mode must be informed when entering and exiting function calls. By doing so, the temporaries are effectively reclaimed in a stack-based manner. The operation, `mark_temp_stack` is used to indicate the initiation of function calls. A function that returns a nested sequence, must insure on completion, that the result is pushed onto the top of the temporary stack. This is accomplished by use of the `keep_temp` operation. Examine example program 3 in appendix B for full details.

5. Planned Enhancements

The initial version of the Data Parallel Library used a primitive two stack memory management system and had only a few operations implemented. The current version has a heap-based memory management system (described in appendix B) and significantly more nested sequence operations implemented. Further improvements fall into three categories. The first is the implementation of new nested sequence operations. Though the library is functional in its current state, there are additional operations that, if implemented, could make the transformations simpler and better. These operations will be implemented and added to the

library to support the transformations. The second is improving efficiency of existing operations through optimizations. Much of the initial implementation of the library has been in the "get things working" context. Once things are working, it will become apparent that we would like them to work better. These improvements will be done on an "as needed" or "as discovered" basis. One area that will be examined is attaining a higher level of reuse of existing vectors. Preliminary tests indicate that certain operations generate multiple copies of vectors rather than always reusing them. Another efficiency improvement is to look at short circuiting some operations when certain outcomes are determined prior to expensive calculations.

The third category is adding significant new capabilities to the system. The primary addition we are planning is including a new family of operations that preserve locality information and lead to much better execution of regular multi-dimensional structures.

6. Changes

Explicit and anonymous temporary variable reclamation modes -- described in section 4

Removal of initial '`b`' from '`_i`', '`_r`' and '`_b`' suffixes.

Renaming of `range_s` to `range_i` for consistency with other operations.

7. Acknowledgements

My advisor, Jan Prins, and I had numerous useful discussions about all aspects of the library. Without his help and direction, it would not have been realized. Lars Nyland, Rik Faith and Stephen Westfold also contributed through technical discussions, and editorial suggestions. All these efforts are greatly appreciated.

A. Listing and Examples of DPL Sequence Operations

Note that the '+' superscript means that one or more of these parameters are expected.

BASE

Description: Offsets the indexing of a sequence by the integer value provided. Makes no direct changes to the values of the sequence input.

Function versions with signatures:

```
base_s  Seq(α) × Int → Seq(α)
base_p  Seq(Seq(α)) × Seq(Int) → Seq(Seq(α))
```

Example applications:

```
base_s([0,2,3,5,7],-5)
       yields: [0,2,3,5,7] with the index of 0 as -5, 2 as -4, etc.
base_p([[2,3],[0],[1,7,5,9,8]],[6,34,-4])
       yields: [[2,3],[0],[1,7,5,9,8]]
       with indices of 2 as 6, 3 as 7, 0 as 34, 1 as -4, etc.
```

BUILD

Description: Creates a sequence from an arbitrary number of component values

Function versions with signatures:

```
build_i     Int × Int+ → Seq(Int)
build_r     Int × Real+ → Seq(Real)
build_b     Int × Bool+ → Seq(Bool)
build_s     Int × Seq(α)+ → Seq(Seq(α))
build_p     Int × Seq(α)+ → Seq(Seq(α))
```

Example applications:

```
build_b(10,T,F,F,T,F,T,F,T,T,T)
       yields: [T,F,F,T,F,T,T,T]
build_s(4,[1,2],[3,4,5],[6],[7,8,9,0])
       yields: [[1,2],[3,4,5],[6],[7,8,9,0]]
build_p(4,[1,2,3],[1,2,3],[1,2,3],[1,2,3])
       yields: [[1,1,1,1],[2,2,2,2,2],[3,3,3,3]]
```

COMBINE

Description: Interleaves two sequences based on a boolean sequence to form a single sequence.

Function versions with signatures:

```
combine_s    Seq(Bool) × Seq(α) × Seq(α) → Seq(α)
combine_p    Seq(Seq(Bool)) × Seq(Seq(α)) × Seq(Seq(α))→ Seq(Seq(α))
```

Example applications:

```
combine_s([T, T, F, F, T, T, T, F],[0,2,3,5,7],[1,4,6])
       yields: [0,2,1,4,3,5,7,6,8,9]
combine_s([F, T, F],[[3,7]],[[1,2],[3,6]])
```

```
        yields: [[1,2],[3,7],[3,6]]
combine_p([[F,F],[T,F,F,T,F]],[[],[0,3]],[[1,7],[5,9,2]])
        yields: [[1,7],[0,5,9,3,2]]
```

COUNT TRUES

Description: Counts the number of true values in a boolean sequence. Used to more efficiently implement conditional operations.

Function versions with signatures:

```
count_trues_s  Seq(Bool) → Int
count_trues_p  Seq(Seq(Bool)) → Seq(Int)
```

Example applications:

```
count_trues_s([T,T,F,T,F,F,F,T])
        yields: 4
count_trues_p([[T,T,F],[F,F],[T,T,F,T,T,F,T])
        yields: [2,0,5]
```

CREATE EMPTY

Description: Generates empty sequences.

Function versions with signatures:

```
cr_empty_s      Int → Seq(α)
cr_empty_p      Int × Int → Seq(Seq(α))
```

Example applications:

```
cr_empty_s(BOOL_type)
        yields: [] an empty boolean sequence
cr_empty_p(FLPT_type, 3)
        yields: [[[[]]]] an empty floating point sequence
```

DISTRIBUTE

Description: Replicates input values a specified number of times creating a nested sequence.

Function versions with signatures:

```
dist_i  Int × Int →  Seq(Int)
dist_r  Real × Int →  Seq(Real)
dist_b  Boolean × Int →  Seq(Bool)
dist_s  Seq(α) × Int → Seq(Seq(α))
dist_p  Seq(α) × Seq(Int) → Seq(Seq(α))
```

Example applications:

```
dist_i(5,8)
        yields: [5,5,5,5,5,5,5,5]
dist_s([3,1,7],3)
```

```
        yields: [[3,1,7],[3,1,7],[3,1,7]]
dist_s([[1,2],[3,6,7]],2)
        yields: [[1,2],[3,6,7]],[[1,2],[3,6,7]]]
dist_p([1.2,3.7,-2.6],[3,1,5])
        yields: [[1.2,1.2,1.2],[3.7],[-2.6,-2.6,-2.6,-2.6,-2.6]]
```

EMPTY

Description: Determines if a sequence is empty or not.

Function versions with signatures:

```
empty_s     Seq(α) → Bool
empty_p     Seq(Seq(α)) → Seq(Bool)
```

Example applications:

```
empty_s([])
        yields T
empty_p([[],[2,3,4],[1,7],[],[]])
        yields: [T,F,F,T,T]
```

INDEX (see also mke_idx and app_idx)

Description: Performs multi-level, sequence-value extractions. Requires auxiliary operations mke_idx and app_idx to create "index blocks". Index blocks are multi-level sets of indices that allow many sequence elements to be extracted in parallel.

Function versions with signatures:

```
index_i     Seq^k(Int) × Seq(Int)_1 × … × Seq(Int)_k → Int
index_r     Seq^k(Real) × Seq(Int)_1 × … × Seq(Int)_k → Real
index_b     Seq^k(Boolean) × Seq(Int)_1 × … × Seq(Int)_k → Bool
index_s     Seq^k(α) × Seq(Int)_1 × … × Seq(Int)_d → Seq^(k-d)(α) where k > d
index_p     Seq(Seq^k(α)) × Seq(Int)_1 × … × Seq(Int)_d) → Seq(Seq^(k-d)(α))
                                                          where k ≥ d
```

Example applications:

```
index_i([0,7,3,4,9,2],[5]);
        yields: 9
index_i([[0,9,6],[4,3,2,8,1],[7]],[2],[3]);
        yields: 2
index_s([[1,2],[3],[4,7,5],[9,1]],[4,2,3,3]);
        yields: [[9,1],[3],[4,7,5],[4,7,5]]
index_p([[1,3,7,2,5],[0,1,2],[4,3,8],[4,7,6,0,3]],[1,2,3,4],[4,2,3,3]);
        yields: [2,1,8,6]
index_p([[[1,3,2,5],[0,1,2],[3,8]],[[4,7,1,3],[5,6,8,3]]],[2,1,1,2],[2,3,2,2]);
        yields: [[5,6,8,3],[3,8],[0,1,2],[5,6,8,3]]
```

LENGTH

Description: Returns the length of a sequence

Function versions with signatures:

```
length_s      Seq(α) → Int
length_p      Seq(Seq(α)) → Seq(Int)
```

Example applications:

```
length_s([2.3,4.5,-12.3,0.5,6.4])
       yields: 5
length_p([[F,F],[T,F,F,T,T],[T,T,T,F],[]])
       yields: [2,5,4,0]
```

PARTITION

Description: Splits a sequence into two parts as evenly as possible.

Function versions with signatures:

```
partition_s   Seq(α) → Seq(Seq(α))
partition_p   Seq(Seq(α)) → Seq(Seq(Seq(α)))
```

Example applications:

```
partition_s([2,3,4,5,6])
       yields: [[2,3,4],[5,6]]
partition_p([[1,2,3,4,5],[2,4,6,8],[3]])
       yields: [[[1,2,3],[4,5]],[[2,4],[6,8]],[[3],[]]]
```

RANGE (see also range origin 1 and stepped range)

Description: Creates an increasing sequence by enumerating all the integers between the lessor first value and the greater second value. If the second value is less than the first, the empty sequence is returned.

Function versions with signatures:

```
range_i       Int × Int → Seq(Int)
range_p       Seq(Int) × Seq(Int) → Seq(Seq(Int))
```

Example applications:

```
range_i(4,16)
       yields: [4,5,6,7,8,9,10,11,12,13,14,15,16]
range_i(7,3)
       yields: []
range_p([-5,7,6,0],[0,7,2,8])
       yields: [[-5,-4,-3,-2,-1,0],[7],[],[0,1,2,3,4,5,6,7,8]]
```

RANGE ORIGIN 1

Description: Shorthand notation for calling range with the first value of 1. All inputs less than or equal to zero yield the empty sequence.

Function versions with signatures:

```
range1_s      Int → Seq(Int)
range1_p      Seq(Int) → Seq(Seq(Int))
```

Example applications:

```
range1_i(7)
      yields: [1,2,3,4,5,6,7]
range1_p([3,7,0,5])
      yields: [[1,2,3],[1,2,3,4,5,6,7],[],[1,2,3,4,5]]
```

REDUCTION

Description: Performs reduction by an arbitrary associative operation. Both a single and parallel version of the reduction operation must be supplied. See the mke_red operation.

Function versions with signatures:

```
reduce_i      (Func(Int × Int) → Int) × Seq(Int) → Int
reduce_r      (Func(Real × Real) → Real) × Seq(Real) → Real
reduce_b      (Func(Bool × Bool) → Bool) × Seq(Bool) → Bool
reduce_s      (Func(Seq(α) × Seq(α)) → Seq(α)) × Seq(Seq(α)) → Seq(α)
reduce_p      (Func(α × α) → α) × Seq(Seq(α)) → Seq(α)
```

Example applications:

```
reduce_i(plus,[2,3,4,5,6])
      yields: 20
reduce_r(mult,[2.34,6.04,14.2])
      yields: 200.697
reduce_p(plus,[[2,7,19],[7,9,12,6],[5,16,-17])
      yields: [28,34,4]
```

RESTRICT

Description: Filters an input sequence with an input boolean sequence.

Function versions with signatures:

```
restrict_s    Seq(Bool) × Seq(α) → Seq(α)
restrict_p    Seq(Seq(Bool)) × Seq(Seq(α)  → Seq(Seq(α))
```

Example applications:

```
restrict_s([T,F,F,T,T,F,T,T,F],[4,1,6,7,2,5,0,9,8])
      yields: [4,7,2,0,9]
restrict_s([T,F,T],[[4,5],[6,0,3,1,2],[9,1]])
      yields: [[4,5],[9,1]]
```

```
restrict_p([[F,F],[T,F,F,T,T],[T,T]],[[4,5],[6,0,3,1,2],[9,1]])
        yields: [[],[6,1,2],[9,1]]
```

SPLIT

Description: Splits a sequence based on a supplied boolean sequence

Function versions with signatures:

```
split_s     Seq(Bool) × Seq(α) → Seq(Seq(α))
split_p     Seq(Seq(Bool)) × Seq(Seq(α)) → Seq(Seq(Seq(α)))
```

Example applications:

```
split_s([T,F,F,T,T,F,F,T,F,F],[0,1,2,3,4,5,6,7,8,9])
        yields: [[0,3,4,7],[1,2,5,6,8]
split_s([T,F,F,T],[[1,2],[3,4,5],[6],[7,8,9,0]])
        yields: [[[1,2],[7,8,9,0]],[[3,4,5],[6]]]
split_p([[T,T,T],[T,F,F,F,T],[F,F]],[[1,2,3],[9,8,7,6,5],[0,1]])
        yields: [[[1,2,3],[]],[[9,5],[8,7,6]],[[],[0,1]]
```

STEPPED RANGE

Description: Creates sequences that can be either increasing or decreasing and allows indirect specification of a step value between consecutive elements of the sequence. This is accomplished by specifying the first and second elements of the sequence, the step is computed as the difference.

Function versions with signatures:

```
st_range_i    Int × Int × Int → Seq(Int)
st_range_r    Real × Real × Real → Seq(Real)
st_range_p    Seq(Num) × Seq(Num) × Seq(Num) → Seq(Seq(Num))
```

Example applications:

```
st_range_i(3,5,11)
        yields: [3,5,7,9,11]
st_range_r(5.34,4.23,0)
        yields: [5.34,4.23,3.12,2.01,0.9]
st_range_p([2,7,5],[7,9,0],[22,16,-23])
        yields: [[2,7,12,17,22],[7,9,11,13,15],[5,0,-5,-10,-15,-20]]
```

B. Sample C programs using DPL

This appendix provides three examples of C programs using the DPL operations. They are illustrative in nature and, in many instances, unnecessarily verbose.

EXAMPLE 1: the odd squares generation routine

The first example is a code segment that would be called from a larger program. It generates a sequence of the squares of all the odd numbers from 1 to the input value. This version of the program employs explicit reclaimation of sequence memory.

```
nseq oddsqs(int n)
{
/* This function returns a sequence of the squares of all the odd */
/* numbers between 1 and n */

nseq   A, B, C, D, E;

    A = range1_s(n);         /* generates the numbers 1,2,3, ... n */
    B = dist_i(2,n);         /* makes a sequence of 2's, same length as A */
    C = mod_p(A,B);          /* makes a boolean flag to mask even numbers */
    fre_nseq(B);             /* releases B vector when it is no longer needed */
    D = restrict_s(C,A);     /* removes even numbers from the list 1 to n */
    fre_nseq(A);             /* releases A vector */
    fre_nseq(C);             /* releases C vector */
    E = mult_p(D,D);         /* squares the odd numbers */
    fre_nseq(D);             /* releases D vector */
    return E;                /* returns sequences containing the squares */
}
```

Here is another version of the same program that uses the anonymous reclaimation approach.

```
{
nseq   A, B;
    mark_temp_stack();
    A = assign(restrict_s(mod_p(range1_s(n)),range1_s(n)));
    B = assign(mult_p(A,A));
    fre_nseq(A);
    return keep_temp(B);
}
```

EXAMPLE 2: the three-dimensional index summation program

The second example is a complete program that can be compiled and run as written. It generates a nested sequence of the sums of the indices of a partially filled three dimensional array. The pure C program that generates the same values is:

```
    for (i=1;i<=n;i++){
        for (j=1;j<=i,j++){
            for (k=1;k<=j;k++){
                rslt[i][j][k] = i+j+k;
            }
        }
    }
```

The C + DPL program requires only enough memory to store the results, it does not need to declare an n-cubed array before beginning the calculation. This example also illustrates explicit reclaimation

```
#include <stdio.h>
#include <dpl.h>
nseq sum_indices_3D(int s)
{
/* This func yields a sequence of summations of indices of a partially filled */
/* 3-D structure.  Entries only exist where subsequent indices are less */
/* than or equal to previous indices.  For example: 4,3,1 and 2,2,2 are valid, */
/* but 2,4,1 and 4,1,2 or not. */

nseq   A, B, C, D, E, F, G, H, I, J, K, L, M, N;

    A = range1_s(s);      /* generates the numbers 1,2,3, ... s */
    B = range1_p(A);      /* forms a sequence [[1],[1,2],[1,2,3]...[1,2,3...s]] */
    C = extract(B,1);     /* flattens the sequence to [1,1,2,1,2,3,..s...] */
    fre_nseq(B);
    D = dist_p(A,A);      /* forms a sequence [[1],[2,2],[3,3,3]...[s,s...s]] */
    fre_nseq(A);
    E = extract(D,1);     /* flattens sequence to [1,2,2,3,3,3,...,s...] */
    F = dist_p(E,C);      /* forms a sequence [[1],[2],[2,2],[3],[3,3],[3,3,3]...*/
    fre_nseq(E);
    G = extract(F,1);     /* flattens sequence to [1,2,2,2,3,3,3,3,3,3,...s...] */
    fre_nseq(F);
    H = dist_p(C,C);      /* forms a sequence [[1],[1],[2,2],[1],[2,2],[3,3,3]...*/
    I = extract(H,1);     /* flattens sequence to [1,1,2,2,1,2,2,3,3,3,...s...] */
    fre_nseq(I);
    J = plus_p(G,I);      /* adds first two sets of indices */
    fre_nseq(G);
    K = range1_p(C);      /* forms a sequence [[1],[1],[1,2],[1],[1,2],[1,2,3]...*/
    fre_nseq(C);
    L = extract(K,1);     /* flattens sequence to [1,1,1,2,1,2,1,2,3,....s...] */
    fre_nseq(K);
    M = plus_p(J,L);      /* adds sum of first two indices and the third index */
    fre_nseq(J);
    fre_nseq(L);
    N = insert(M,H,1);    /* shapes the result into desired structure */
    fre_nseq(H);
    fre_nseq(M);
    O = insert(N,D,1);    /* completes shaping of result into desired form */
    fre_nseq(D);
    fre_nseq(N);
    return O;             /* returns nested sequence containing the result */
}
void main();
{   nseq rslt;
    setmem();
    rslt = sum_indices_3D(4);
    show_p(rslt);
}
```

EXAMPLE 3: the median calculation using recursion

The third example program performs finds the median element in a sequence. It is more complex than the other two examples and is provided only for users who would actually use the library for computational purposes. Though complicated, it is supplied for three reasons. First it provides a useful template of a non-trivial DPL program illustrating by example how many different features of the library work together

and second provides an example of output code that is generated by the Proteus transformation system. Finally, it provides a realistic example of the anonymous reclamation mode.

```
#include <stdio.h>
#include <dpl.h>
int kth_item(nseq S, int k)
{
/* Uses divide and conquer to recursively determine the kth element in a*/
/* list.  Randomly selects a pivot and splits the list into those that are */
/* greater than the pivot, those that are less and those that are equal.*/
/* Calculates which list the "kth" element must be in, recursively applies */
/* the routine to the new list with the new element position */

mark_temp_stack();
    {int pivot = random_i(S);                          /* randomly select pivot */
    nseq P = assign(dist_i(pivot, length_s(S)));       /* spread copies of pivot */
    nseq lower = assign(restrict_s(less_p(S, P), S)); /* compare each value w/ pivot */
    int len1 = length_s(lower);             /* count number less than pivot */
    nseq same = assign(restrict_s(equal_p((S, P), S)); /* count # equal to the pivot */
    int result;
    if (len1 >= k){                         /* if there are k or more elts less, */
        fre_nseq(P);                        /* recursively apply kth-item to the */
        fre_nseq(same);                     /* list of lower elements */
        result = assign(kth_item(lower, k));
        fre_nseq(lower);
    else{
        {fre_nseq(lower);
        int len2 = length_s(same) + len1;   /* if kth item is in equal list */
        fre_nseq(same);                     /* simply return pivot value */
        if (len2 >= k){
            result = dist_i(pivot,1);}
        else{                               /* else do kth-item to greater */
            result = assign(kth_item(restrict_s(grt_p(S, P),
                        S), k - len2));}  /* list with k adjusted */
            fre_nseq(P);}}
    return keep_temp(result);}}

int median(nseq S)                                     /* the median is found by */
{
mark_temp_stack();
    {int result = assign(kth_item(S, length_s(S)/2)); /* calling kth-item w/ k */
    clear_temp();
    return result;}}                                   /* = to half the listsize */

void main()
{   setmem();
    printf("Median is : %d\n",median(range1_s(1000000)));
}
```

C. Implementation Issues

The implementation of DPL in terms of vector operations supplied by CVL relies on an effective representation of nested sequences with vectors. There are several different schemes, but the one we have selected allows simple detachment and re-attachment of the nesting structures. A nested sequence of depth d is represented with a set of d + 1 vectors. The first *d* vectors are descriptor vectors. Each descriptor vector

defines the number and size of the sub-sequences of the next subsequent level. The remaining vector is the value vector and contains the actual base-type values of the sequence.

A nested sequence, [[2, 9, 0], [7, 8, 0, 2], [9]], is represented using three vectors. The two descriptor vectors are: [3] to indicate that at the highest level there are three elements in the nested sequence, and [3, 4, 1] to indicate that the three elements are themselves sequences with the respective lengths 3, 4, and 1. The value vector in this case is [2, 9, 0, 7, 8, 0, 2, 9]. Given this representation, it is easy to see how the insert and extract operations are implemented.

Many of the sequence operations are implemented by operating on the top descriptor vector and propagating the changes to the next level of descriptors. The cumulative effect is then applied to the value vector, yielding the result of the sequence operation.

Because of widely varying support from the operating systems available on parallel machines, CVL requires that all vector memory to be used in an entire program be allocated once, in a single, monolithic block. All memory management operations must be handled by the calling program and must work within the constraint of the original allocation block. DPL provides all memory management operations necessary to hide CVL's primitive support from calling programs. DPL operates the single memory block as a heap, automatically allocating new vectors when the calling program performs an operation that yields a new nested sequence. To make the best use of available memory as possible, DPL uses a reference counting scheme to reuse vectors whenever possible. Often, nested sequences will have many of the same descriptor vectors. Instead of allocating multiple vectors with the same values, DPL uses an indirection structure to allow safe reuse of vectors. When a vector is reclaimed by the calling program, DPL checks to ensure that the actually deallocation is only performed if there are no outstanding references to the vector. When the entire monolithic block of vector memory is used, and a new request cannot be satisfied, DPL initiates a garbage collection phase to de-fragment memory and attempt to honor the request.

Bibliography

[BCS+90] Blelloch, G., Chatterjee, S., Sipelstein, J., Zahga, M., "CVL: A C Vector-Library", Draft Technical Note, Carnegie Mellon University, 1990.

[Blel90] Blelloch, G., *Vector Models for Data-Parallel Computing,* MIT Press, 1990.

[Blel92] Blelloch, G., “NESL: A Nested Data-Parallel Language version 2.6”, Technical Report CMU-CS-92-129, Carnegie Mellon University, April 1993.

[BS90] Blelloch, G., Sabot, G., "Compiling Collection-Oriented Languages onto Massively Parallel Computers", *Journal of Parallel and Distributed Computing*, 8(2), February 1990.

[CBZ90] Chatterjee, S., Blelloch, G., Zagha, M., "Scan Primitives for Vector Computers", *Proceedings Supercomputing '90*, IEEE , 1990.

[FHS91] Faith, R., Hoffman, D., Stahl, D,"UnCVL: MasPar Implementation of the CVL Library”, Technical Report UNC-TR93-063, University of North Carolina - Chapel Hill, 1993.

[MNP+91] Mills, P., Nyland, L., Prins, J., Reif, J., Wagner, R.,"Prototyping Parallel and Distributed Programs in *Proteus*", *Proceedings Symposium on Parallel and Distributed Processing 92*. 1992.

[MNP+92] Mills, P., Nyland, L., Prins, J., Reif, J., "Prototyping Parallel and Distributed Programs in *Proteus*", *Proceedings Symposium on Parallel and Distributed Processing 92.* 1992

[PP91] Prins, J., Palmer, D., "Transforming High-Level Data-Parallel Programs into Vector Operations”, *Proceedings Principles and Practices of Parallel Programming 93*, ACM, 1993.