

Dartmouth College Computer Science

Technical Report PCS-TR95-250



DartCVL: The Dartmouth C Vector Library

Thomas H. Cormen,¹ Sumit Chawla,² Preston Crow,³
Melissa Hirschl,² Roberto Hoyle, Keith D. Kotay,²
Rolf H. Nelson,⁴ Nils Nieuwejaar,⁵ Scott M. Silver,
Michael B. Taylor, Rajiv Wickremesinghe

Dartmouth College
Department of Computer Science

Abstract

As a class project, we implemented a version of CVL, the C Vector Library, on a DECmpp 12000/Sx 2000, which is equivalent to the MasPar MP-2 massively parallel computer. We compare our implementation, DartCVL, to the University of North Carolina implementation, UNCVL.

DartCVL was designed for the MP-2 architecture and UNCVL was designed for the MP-1. Because the MasPar MP-1 and MP-2 are functionally equivalent, both DartCVL and UNCVL will run on either. Differences in the designs of the two machines, however, may lead to different software design decisions. DartCVL differs from UNCVL in two key ways. First, DartCVL uses hierarchical virtualization, whereas UNCVL uses cut-and-stack. Second, DartCVL runs as much serial code as possible on the console, whereas UNCVL runs all serial code on the Array Control Unit (ACU). The console (a DECstation 5000/240 at Dartmouth) has a significantly faster serial processor than the ACU.

DartCVL is optimized for the MP-2, and our timing results indicate that it usually runs faster than UNCVL on the 2048-processor machine at Dartmouth.

Authors' address (unless otherwise stated): 6211 Sudikoff Laboratory, Hanover, NH 03755. Send electronic mail inquiries to Tom Cormen, thc@cs.dartmouth.edu.

¹Supported in part by funds from Dartmouth College and in part by the National Science Foundation under Grant CCR-9308667.

²Supported by a Dartmouth College Graduate Fellowship.

³Supported by NASA Graduate Student Researchers Program Fellowship NGT-51160.

⁴Currently with Digital Equipment Corporation. Work performed while at Dartmouth College.

⁵Supported in part by the NASA Ames Research Center under Agreement Number NCC 2-849.

1 Introduction

As a class project, the authors implemented CVL, the C Vector Library, on a DECmpp 12000/Sx 2000. CVL is an interface defined by Blelloch et al. [BCH⁺93] to a group of simple functions for managing and operating on vectors. Our implementation, DartCVL, embodies the many functions defined in [BCH⁺93]. The target machine is equivalent to the MasPar MP-2, a massively parallel SIMD computer.

We had two goals in this project. The educational goal was to give the students (undergraduate and graduate students at Dartmouth) experience in programming a massively parallel machine and an understanding of the tradeoffs in ease of programming and performance. The engineering goal was to produce a fast implementation of CVL.

This paper focuses on the engineering goal. We were aware of the UNCVL project [FHS93], which is an implementation of CVL for the MasPar MP-1. Although the MP-1 and MP-2 are code-compatible, their architectures are sufficiently different that design decisions that work well on one machine may not perform as well on the other. Consequently, some of the fundamental design decisions we made for DartCVL differ from those made for UNCVL. DartCVL benefitted from these differences in most, but not all, cases.

CVL

The best pocket description of CVL comes from its manual [BCH⁺93]:

CVL is a library of low-level vector routines callable from C. This library presents an abstract model of a vector machine suitable either for stand-alone use or as the backend of a high-level language system. CVL includes a rich set of vector operations including both elementwise computations, and more global operations such as scans, reductions, and permutations. The library also includes segmented versions of these global operations; segmented operations are crucial for the implementation of nested data-parallel languages.

CVL vectors may be of any nonnegative length, and segmented vectors may have any segmentation that conforms to the length. Scalars are indistinct from vectors in CVL; that is, a scalar is simply a vector of length 1.

All CVL function names are three letters followed by an underscore followed by three more letters, e.g., `add_wuz`. Each name has four components:

1. The first three letters are a mnemonic for the root function to be applied, e.g., `add` (addition), `sub` (subtraction), and `cpy` (copy).
2. The first letter following the underscore is a consonant denoting the class the CVL function belongs to, e.g., `w` (elementwise), `s` (scan), and `p` (permute).
3. The second letter following the underscore is a vowel indicating the kind of vector to which the function is applied; the choices are `u` (unsegmented), `e` (segmented), and `o` (non-vector operation).
4. The third letter following the underscore is a consonant giving the type of the individual elements, e.g., `z` (integer), `b` (boolean), `d` (double), and `s` (segment descriptor).

Thus, the CVL function `add_wuz` performs elementwise addition on unsegmented vectors of integers.

Most parameters to CVL functions are either integers giving vector lengths or numbers of segments, or of the type `vec_p`, which is an abstract handle for accessing vector memory. In DartCVL, the `vec_p` type is defined by

```
typedef plural void *vec_p;
```

which, in MPL¹ terminology, is a “singular pointer to plural.” That is, a `vec_p` is a pointer to the same location across the local memories of all the processing elements. The CVL function `add_wuz` has the prototype

```
void add_wuz(vec_p d, vec_p s1, vec_p s2, int len, vec_p scratch);
```

whose parameters are a handle `d` for the result (or destination) of the elementwise vector addition, handles `s1` and `s2` for the source vectors, the number `len` of elements in either of the operands or the result, and a handle `scratch` to scratch space, should it be needed to perform the function.

CVL functions are divided into seven classes, each of which has an associated letter that follows the underscore in function names:

elementwise (w): Perform an operation on every element of the vector operands.

reduce (r): Combine all elements of a vector together under an associative function such as addition or maximum.

scan (s): Create a vector whose i th element is the reduction of the first $i - 1$ elements of the operand.

permute (p): Rearrange the elements of a vector according to an index vector.

vector-scalar (v): Convert vectors to scalars and vice versa.

facilities (f): Perform needed system functions and create vectors and segment descriptors.

library (l): Functions that may be implemented in terms of other CVL functions.

Outline

The remainder of this paper is organized as follows. Section 2 describes the DECmpp 12000/Sx 2000 architecture and its effect on some of the design decisions we made in DartCVL. Section 3 discusses some other design decisions. Section 4 presents and analyzes timing results of DartCVL and UNCVL functions. Finally, Section 5 presents some concluding remarks.

2 Machine architecture

In this section, we describe the DECmpp 12000/Sx 2000 [Dig92], which is identical to the MasPar MP-2. For concrete examples, we will use the 2048-processor machine (named “cascade”) installed at Dartmouth.

The DECmpp 12000/Sx 2000 is a massively parallel processing system, made up of a *console system* and a *data parallel unit* (DPU). The console is a workstation providing standard I/O devices. For cascade, the console system is a DECstation 5000/240.

¹MPL is the programming language based on C used to program the MasPar MP-1 and MP-2. It is tied to the machine architecture described in Section 2.

The DPU is made up of an *array control unit*, or *ACU*, an array of *processor elements*, or *PEs*, and a PE communication system. The number of PEs is a power of 2 in the range 1K to 16K. We denote the number of PEs by `nproc`; in cascade, `nproc = 2048`. The ACU is a serial processor in its own right, and it acts as a controller for the PE array. All parallel processing takes place within the DPU.

Code can execute in any one of three places:

- All code operating on parallel (or, in MPL terminology, *plural*) data executes on the processors of the PE array.
- Code operating on *singular* (i.e., non-parallel) data within modules of parallel code executes on the ACU.
- Modules of singular code may execute on the console.

The console is significantly faster than the ACU, and the ACU is significantly faster than the individual PEs. The MPL programming environment includes library functions to copy data between the console and the ACU and between the console and the PE array; these copying functions incur a slight overhead. In general, therefore, long stretches of singular code are best run on the console, short stretches of singular code within parallel code are best run on the ACU, and inherently parallel code is best run within the PE array.

All CVL functions are called from the console. Both DartCVL and UNCVL implementations call functions in the DPU as necessary to run parallel code. Most DartCVL functions compute some simple scalars in the console prior to calling DPU functions.

Each PE has its own processor and data memory; cascade has 64K bytes per PE, which is the maximum supported by the DECmpp 12000/Sx 2000 architecture. When the ACU sends an instruction to the PEs, each PE carries it out only on data that reside physically in that PE. If a computation requires data from two or more PEs, the PE communication system must send the data to a common PE so that the PE can perform the operation.

The DECmpp 12000/Sx 2000 contains two networks for routing data among the PEs:

- The *X-Net* connects each PE to its eight immediate neighbors in a two-dimensional mesh in which the PEs are arranged.
- The *global router* routes data in arbitrary communication patterns.

Although the X-Net is significantly faster than the global router, CVL does not include mesh-oriented permuting functions. DartCVL uses the X-Net whenever possible for its internal operation, but its permuting functions are forced to use the global router.

The DECmpp 12000/Sx 2000 includes an I/O subsystem as well, but it is not used by DartCVL.

Virtualization

When a vector has more elements than there are PEs, each PE acts as a number of *virtual processors*. We can think of each element as having its own virtual processor. If N elements are spread as evenly as possible across the PEs, the PE with the most elements contains $\lceil N/nproc \rceil$, which we refer to as the *virtual processor ratio*, or *VPR*.

Figure 1 shows two ways to organize data under virtual processing. In either case, we can view the data as a two-dimensional array with the column corresponding to the PE number and the row as the offset within the PE. In *cut-and-stack* virtualization, the i th element (indexing from 0)

Cut-and-stack								Hierarchical							
PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7	PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7
0	1	2	3	4	5	6	7	0	3	6	9	12	15	18	
8	9	10	11	12	13	14	15	1	4	7	10	13	16		
16	17	18						2	5	8	11	14	17		

Figure 1: Vector layout of 19 elements on 8 PEs for cut-and-stack and hierarchical virtualization. Blanks indicate unused positions. In both cases, the VPR is 3. With cut-and-stack, each PE has either 2 or 3 elements. With hierarchical, PEs 0 through 5 have 3 elements, PE 6 (the pivot) has 1 element, and PE 7 has none.

resides in PE $i \bmod \text{nproc}$ and has offset $\lfloor i/\text{nproc} \rfloor$. In *hierarchical* virtualization, the i th element has offset $i \bmod VPR$ and resides in PE $\lfloor i/VPR \rfloor$. Cut-and-stack corresponds to row-major layout, and hierarchical corresponds to column-major.

UNCVL uses cut-and-stack virtualization, but we decided to use hierarchical for DartCVL. Cut-and-stack has two advantages. First, it distributes vector elements as evenly as possible across the PEs. For many operations, this even spreading makes no difference, but we found that it helps in permute operations. Second, because the number nproc of processors is a power of 2 and the VPR might not be, positional calculations can be performed slightly faster than with hierarchical virtualization. Hierarchical virtualization has the advantage that scan operations are significantly faster than with cut-and-stack. With cut-and-stack, scan operations are performed row by row. Each row is a scan across all PEs, and so VPR scans across all PEs are required.² With hierarchical, a scan operation is performed by scanning within each PE, then performing just one scan across all PEs to distribute scan information, and then another scan within each PE. Because scans across all PEs are relatively expensive, hierarchical virtualization yields much faster scans than cut-and-stack.

3 DartCVL design decisions

In addition to the design decisions for DartCVL discussed above—using hierarchical virtualization and running certain scalar computations on the console—there were other interesting facets to the DartCVL design. This section discusses some of them.

Vector organization and access

As Figure 1 shows, unlike cut-and-stack, hierarchical virtualization can produce a highly unbalanced load on the PEs. With cut-and-stack, each PE has either VPR or $VPR - 1$ elements of each vector. With hierarchical virtualization, however, some PEs have VPR elements, some have 0 elements, and one PE has between 0 and VPR elements.

To describe vector organization, most of the DartCVL functions that run on the DPU take as an input a structure with three pieces of information:

- the vector’s VPR ,

²Prins [Pri93] reports that a sophisticated pipelined algorithm improves the scan performance with cut-and-stack virtualization.

- the number of the *pivot* PE, which is the one PE with between 0 and *VPR* elements, and
- the number of elements in the pivot PE, which we (inaccurately) call the “last VPR.”

Our original MPL code to step through vector elements used a plural index, as in the following example to perform elementwise integer addition of vectors with base addresses *S1* and *S2* into a result vector with base address *D*:

```

struct vpr_struct {
    int vpr;
    int pivot;
    int lastvpr;
} Vpr;

plural int *S1, *S2, *D;
plural int vpr_per_pe;
plural int i;

vpr_per_pe = (iproc <= Vpr.pivot) ? Vpr.vpr : 0;
proc[Vpr.pivot].vpr_per_pe = Vpr.lastvpr;

for (i = 0; i < vpr_per_pe; i++)
    D[i] = S1[i] + S2[i];

```

Here, *Vpr* is a structure with the three fields described above. The plural integer *vpr_per_pe* holds the number of vector elements in each PE. The plural variable *iproc* is builtin to MPL and contains each PE’s number, between 0 and *nproc* – 1. The *proc[]* construct in MPL indicates an action occurring in just one PE. In this case, we are assigning the value in the *lastvpr* field of the *Vpr* structure to the variable *vpr_per_pe* in the PE whose number is the *pivot* field.

This approach proved to be relatively slow because the for-loop index *i* is plural. Loops with plural conditions entail an implicit test requiring communication among the PEs. In this case, the loop continues iterating as long as any PE has a value of *i* less than its value of *vpr_per_pe*. The machine must execute a global-OR operation to determine if any PE satisfies this condition. We found that this communication exacted a heavy performance cost in an otherwise simple operation.

We devised a faster method by avoiding loops with plural indices. Because plural if-statements in MPL require no communication, we use them instead to mask off PEs once we have exhausted their vector elements. The resulting code is more complex but runs faster. Our final MPL code for elementwise integer addition, which follows, has two further optimizations: putting variables into registers and using pointers rather than array indexing.³

³These optimizations would normally be performed by a good optimizing compiler.

```

struct vpr_struct {
    int vpr;
    int pivot;
    int lastvpr;
} Vpr;

register plural int *s1 = S1, *s2 = S2, *d = D;
register int i;

if (iproc <= Vpr.pivot)
    for (i = 0; i < Vpr.lastvpr; i++, d++, s1++, s2++)
        *d = *s1 + *s2;

if (iproc < Vpr.pivot)
    for (i = Vpr.lastvpr; i < Vpr.vpr; i++, d++, s1++, s2++)
        *d = *s1 + *s2;

```

Fundamentally, cut-and-stack allows slightly faster virtual processor looping, because only one for-loop is required. The following code performs the same elementwise integer addition for cut-and-stack on a vector whose length is given by `len` with just one for-loop:

```

register int steps = len / nproc;
register int leftover = len % nproc;
register int i;
register plural int *s1 = S1, *s2 = S2, *d = D;

for (i = 0; i < steps; i++, d++, s1++, s2++)
    *d = *s1 + *s2;

if (iproc < leftover)
    *d = *s1 + *s2;

```

Segment descriptors

Many CVL operations use segmented vectors, and they are a key part of the implementation of NESL [Ble92], a nested data-parallel language. In a NESL implementation of Quicksort, for example, a single vector is repeatedly segmented into smaller segments, each of which represents a partition of the data. Segmented operations treat each segment as though it were a separate vector.

CVL implementations must have an internal representation of how a vector is segmented. A further complication is that segments may have zero length. (To see why, consider that Quicksort may generate zero-size partitions.) A vector with n elements and m segments may therefore have $m < n$, $m = n$, or $m > n$. CVL defines the functions `mke_fov`, which converts a vector of nonnegative segment lengths into the internal representation, and `len_fos`, which converts the internal representation into a vector of segment lengths.

DartCVL's internal representation for segmentation has three parts, which for a vector with n elements and m segments are as follows:

- A vector of n nonnegative *start counts*. The i th start count is nonzero if and only if the i th position is the first position in a segment. The start count is 1 plus the number of consecutive zero-length segments immediately preceding this segment.

- A vector of n nonnegative *end counts*. The i th end count is nonzero if and only if the i th position is the last position in a segment. The end count is 1 plus the number of consecutive zero-length segments immediately following this segment.
- A vector of m *start indices*. The j th start index is the position in the data at which the j th segment starts.

For example, a vector with segment lengths 0 3 0 0 2 3 would have the following representation:

	index							
	0	1	2	3	4	5	6	7
start count	2	0	0	3	0	1	0	0
end count	0	0	3	0	1	0	0	1
start index	0	0	3	3	3	5		

Only one set of DartCVL functions—segmented reductions—uses the start and end counts as integers. All other DartCVL functions that use the start and end counts only need to know whether they are nonzero; that is, they treat the start and end counts as booleans.

Permuting functions

There is little leeway for optimizing the permuting functions of CVL in MPL. All CVL permuting functions allow arbitrary communication patterns; there is no function exclusively for grid communication, which would permit the implementer to use the X-Net.

The only significant optimization that DartCVL performs in the permuting functions is to communicate via the MPL `router` construct for *VPRs* of 2 or less and the MPL function `sp_rsend()` for *VPRs* of 3 or greater. We found by experimentation that these methods were the best available for these *VPR* ranges. In either case, we send the data row by row to the appropriate destination PEs. Although this method is simple when using `sp_rsend()`, it is a little more complicated when using the `router` construct. The address that the data goes to within the receiving PE is computed within the receiving PE in `sp_rsend()`, but it is computed within the sending PE—which would produce the wrong address—in the `router` construct. When using the `router` construct in DartCVL, we use a nested loop, running through all the receiving rows for each sending row. Because this approach adds significant overhead, we found it to be effective only for *VPRs* of at most 2.

We also tried implementing the permutations as several faster routes through the X-Net rather than one slower call to `sp_rsend()`, but even with conditions as favorable as possible to the X-net method, the `sp_rsend()` call was 30% faster than the required sequence of X-Net calls, and so we abandoned this line of research.

As we shall see in Section 4, DartCVL sometimes permutes slightly slower than UNCVL. When DartCVL is slower, it is typically for lengths just above a multiple of `nproc`. At first, we hoped that hierarchical virtualization would be better in such cases, for the following reason. Consider a vector whose length is one greater than a multiple of `nproc`. Under cut-and-stack, this vector would have $VPR - 1$ elements in PEs 1 through `nproc` - 1 and VPR elements in PE 0. With hierarchical virtualization, the same vector would have VPR elements in PEs 0 through `nproc`/2 - 1 and 1 element in PE `nproc`/2. The first $VPR - 1$ rows send `nproc` elements with cut-and-stack but at most only `nproc`/2 + 1 elements with hierarchical virtualization. The last row sends only 1 element with cut-and-stack and `nproc`/2 with hierarchical. We hoped that each of the first $VPR - 1$ row

sends would be faster with hierarchical, since they infuse fewer elements into the global router. We were wrong; they are not faster. Moreover, the last row send is much faster with cut-and-stack, since it sends only 1 element. Overall, cut-and-stack appears to be a bit faster for permuting.

Miscellaneous features

Here, we note a few miscellaneous features of the DartCVL design.

Boolean representation: CVL includes a boolean type `cvl_bool`, which is up to the CVL implementation to define. At first, we defined `cvl_bool` as a `char` so that it would occupy only one byte. We found certain incompatibilities with this definition, however, so we eventually changed `cvl_bool` to `int`, occupying 32 bits. Although this definition seems to waste space, it works. UNCVL uses essentially the same approach.

Memory efficiency: CVL converts C arrays to CVL vectors via the `c2v_fuz`, `c2v_fub`, and `c2v_fud` functions. For these functions, DartCVL is able to handle larger arrays than UNCVL.

Rank functions: We implemented the ranking functions via Jan Prins’s virtualized bitonic sort.

4 Timing results

To evaluate our implementation of DartCVL, we timed all the CVL functions on cascade (`nproc = 2048`) using both the DartCVL and UNCVL implementations. We achieved our goal—beating the UNCVL running times—in most cases, particularly for large vectors. This section presents a representative subset of our timing results.

We timed each function over a specific set of vector lengths. For each length, we performed ten timing tests, using randomly-generated data in each test. Unfortunately, the timer of the MasPar MP-2 often gives erroneous results. These outliers are so glaring, however, that they are easy to spot and remove, which we did.

The graphs that follow show the ratio of DartCVL to UNCVL running times over selected functions. DartCVL is faster when the ratio is below the dotted line at y -coordinate 1. Each data point represents the average over the ten runs (with outliers removed) for a specific vector length. We tested the set of lengths 1, 1024, 2048, 2049, 3072, 8192, 8193, 9216, 30720, 30721, 31744, 61440, 61441, 62464. (Note that the horizontal point spacing for each function is not proportional to the vector length.) Each length is either an integer multiple of `nproc`, an integer multiple of `nproc` plus `nproc/2`, or an integer multiple of `nproc` plus 1. We expected that DartCVL would perform relatively well on vector lengths that are an integer multiple of `nproc` and not as well on the other lengths. We have graphed ratios only for functions that operate on vectors of integers; results for vectors of doubles or booleans are essentially the same.

Figures 2 and 3 show the relative speeds of DartCVL and UNCVL on scan and reduce functions for addition, maximum, and logical-and. The scan (both unsegmented and segmented) and segmented reduce functions (which calls a segmented scan function) are where we expected the advantages of hierarchical virtualization to be most prominent, and this was indeed the case. For long vectors, scans are over five times faster in DartCVL than in UNCVL, but UNCVL times beat DartCVL for short vectors. For the unsegmented reduce functions, UNCVL is consistently slightly faster than DartCVL. We believe that this behavior is due to virtual processor looping under cut-and-stack being slightly faster than with hierarchical virtualization, as discussed in Section 3.

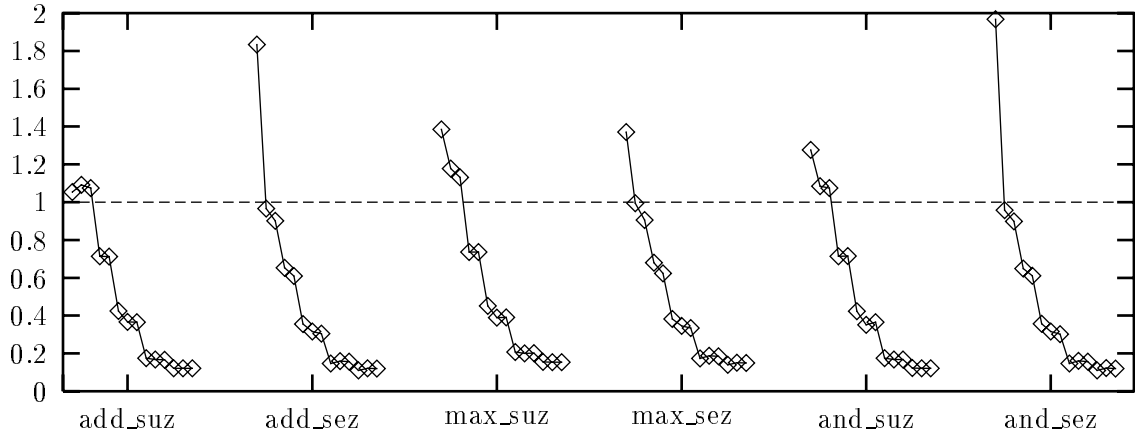


Figure 2: Ratio of DartCVL to UNCVL times for add, max, and logical-and scan functions. In this and all of the following graphs, DartCVL is faster when the ratio is below the dotted line at y -coordinate 1.

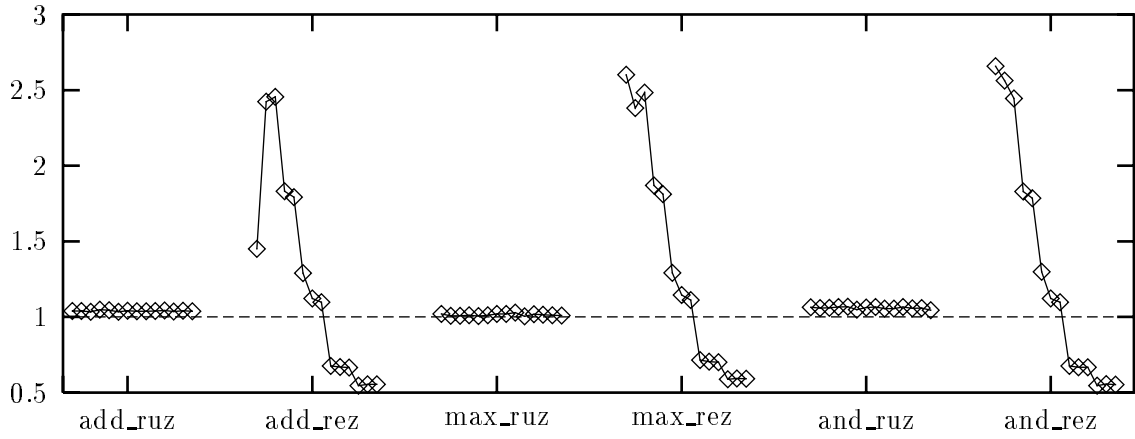


Figure 3: Ratio of DartCVL to UNCVL times for add, max, and logical-and reduction functions.

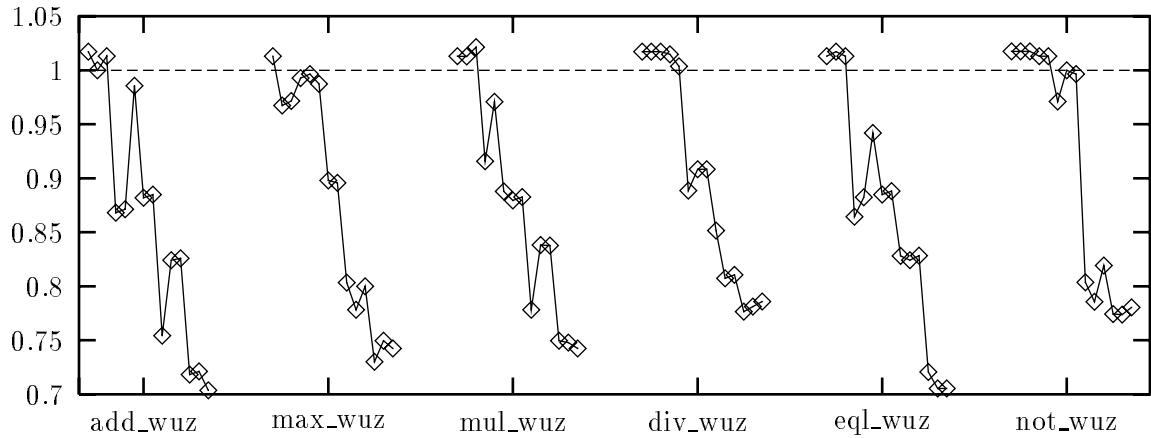


Figure 4: Ratios for add, max, multiply, divide, equality, and logical-not elementwise functions.

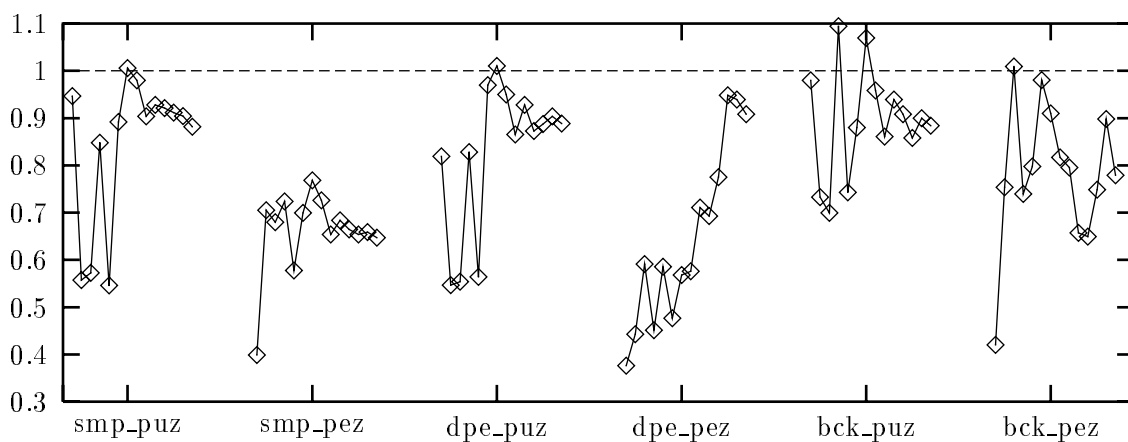


Figure 5: Ratio of DartCVL to UNCVL times for permute functions without flags.

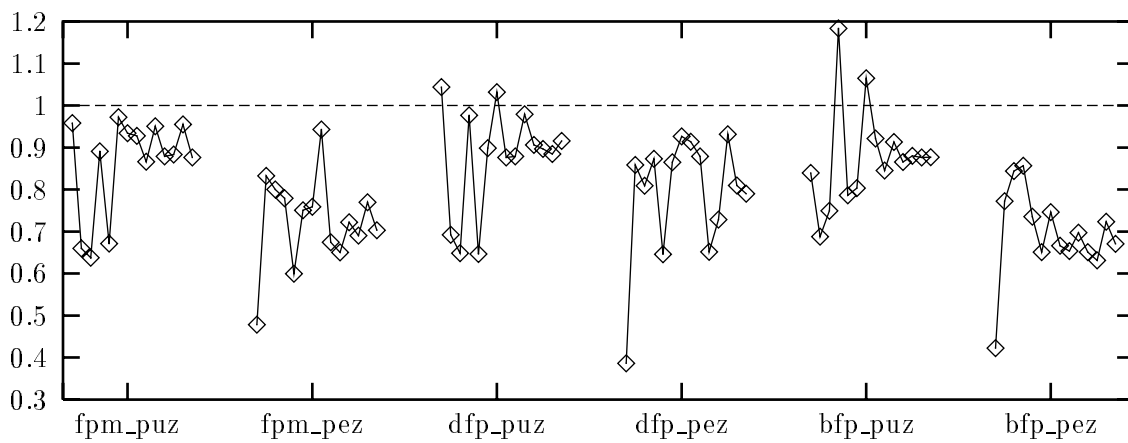


Figure 6: Ratio of DartCVL to UNCVL times for permute functions with flags.

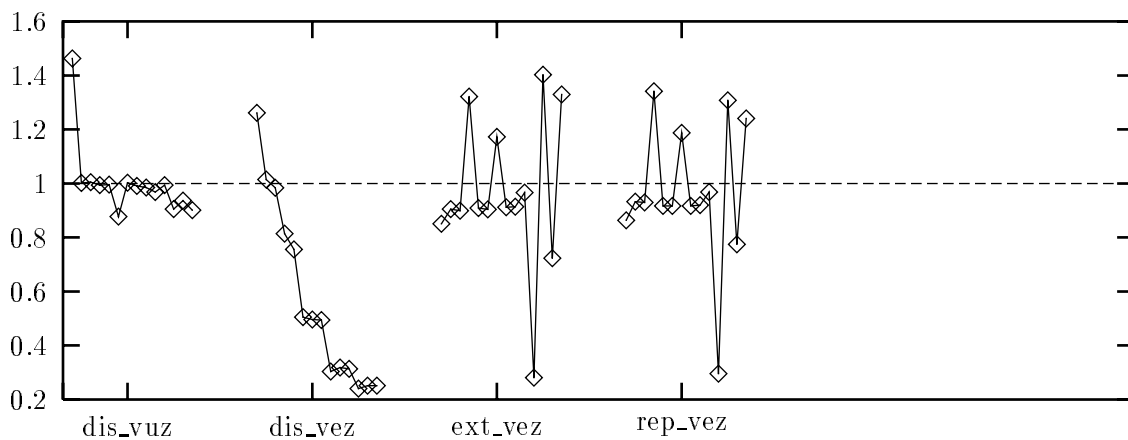


Figure 7: Ratio of DartCVL to UNCVL times for distribute, extract, and replace functions.

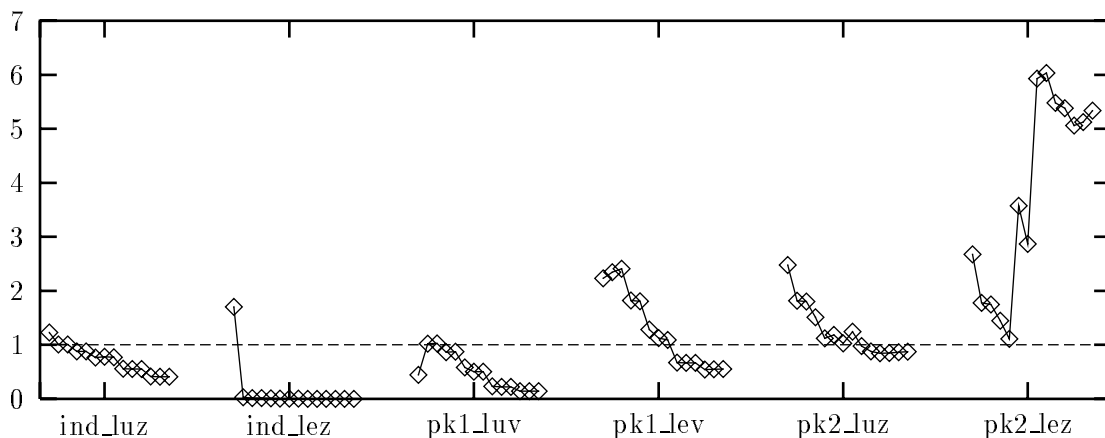


Figure 8: Ratio of DartCVL to UNCVL times for index and pack library functions.

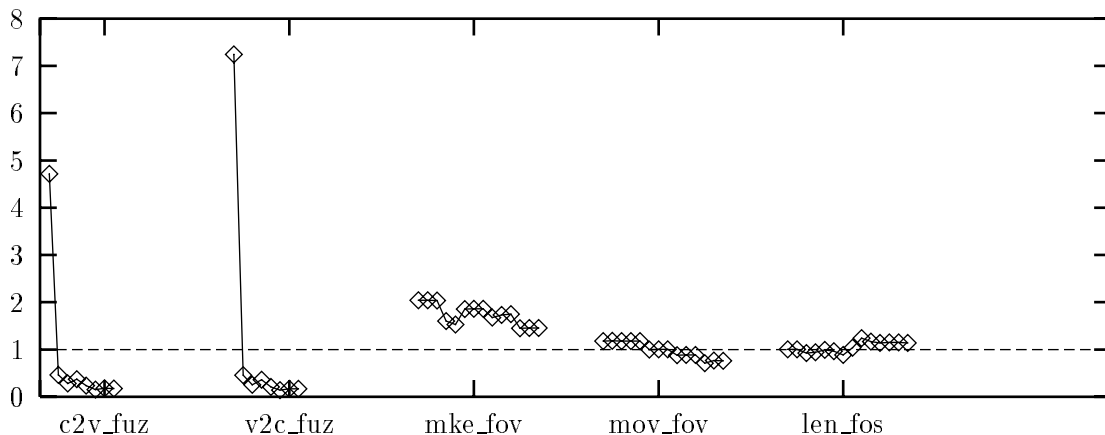


Figure 9: Ratio of DartCVL to UNCVL times for facilities functions to convert between vectors and C arrays, convert between segment descriptors and length vectors, and move vectors.

Figure 4 shows the relative speeds of DartCVL and UNCVL on six elementwise functions. Again, UNCVL is faster for short vectors (we believe due to the extra for-loop overhead in DartCVL) but DartCVL is faster for long vectors. If UNCVL had used the optimizations of DartCVL—register variables and incrementing pointers—it probably would have been consistently faster, although the relative difference would diminish with increasing *VPRs*.

Figures 5 and 6 show the timings for unsegmented and segmented permute functions. Figure 5 includes functions for simple permutes, permutes with default values, and back permutes (i.e., gather operations). Figure 6 includes functions that take flags indicating which elements are to be permuted in simple, default, and back permutes. DartCVL is faster than UNCVL for most, but not all, vector lengths. Almost all cases in which UNCVL beat DartCVL were for vectors lengths equal to one greater than a multiple of `nproc`, as discussed in Section 3.

Figure 7 shows the timings for functions that distribute a scalar over an unsegmented vector, distribute a different scalar to each segment of a vector, extract a scalar from each segment of a vector, and replace a value in each segment of a vector. DartCVL was usually faster than UNCVL,

although `UNCVL` was a bit faster for some vector lengths.

Figure 8 shows the timings for the library functions that create index vectors and pack vectors. Library functions are those that may be implemented solely by calls to other CVL functions but may have alternate, faster implementations. Both `DartCVL` and `UNCVL` implement the unsegmented index function `ind_luz` directly, but `DartCVL` is faster because the for-loops are better optimized. Both implementations perform the segmented index function `ind_1ez` by calls to `dis_vez`, `add_sез`, and `add_wuz`; `DartCVL`'s advantages in these functions give it a marked advantage in `ind_1ez`. The pack functions `pk1_luv` and `pk1_lev` merely return counts of how many items are to be packed, and so they are simply reduce functions to add up nonzero flags. `DartCVL` is faster on segmented vectors for the same reason as for `add_rez`; we are not sure why the behavior for `pk1_luz` differs from the similar `add_ruz`. The functions that actually perform the packing are `pk2_luz` and `pk2_1ez`. `DartCVL` and `UNCVL` implement these functions rather differently. `DartCVL` generates indices using `add_suz` and then calls a flag permute function to move the data. `UNCVL` moves the data directly, performing a monotonic route row by row. For each row, each PE sends at most one item and receives at most one item, and so the routing is particularly fast on the MasPar MP-2.

Finally, Figure 9 shows timings for various facilities functions. Functions `c2v_fuz` and `v2c_fuz`, which convert between C arrays and CVL vectors are quite a bit faster in `DartCVL` due to hierarchical virtualization. `DartCVL` uses the MPL functions `blockIn()` and `blockOut()` to perform aggregate transfer data between the console and PE memories in hierarchical order. `UNCVL` transfers data between the console and ACU memories via the MPL functions `copyIn()` and `copyOut()`, but it also has to serially move the data between the ACU and PE memories. `DartCVL` is faster for large vectors in `mov_fov`, which copies a vector, because it uses the MPL function `p_memcpy()` whereas `UNCVL` copies a row at a time. `UNCVL` is faster than `DartCVL` for the functions `mke_fov` and `len_fos`, which convert between vectors of segment lengths and segment descriptors. We believe this difference is due to the relatively complex structure of the `DartCVL` segment descriptors. We would like to improve these two functions in `DartCVL`, because they are invoked fairly often.

5 Conclusion

We believe that our implementation of `DartCVL` maximizes the advantages of hierarchical virtualization. Moreover, by careful optimization, we have minimized the disadvantages.

We have measured the individual CVL functions in isolation, rather than as they would be executed in actual code. We do not know whether `DartCVL` would be faster than `UNCVL` in actual code generated from, for example, NESL source. We plan to measure which implementation is faster in practice once we assemble a test suite of NESL code.

The `DartCVL` software should become publicly available via anonymous ftp sometime during the spring of 1995. We plan to continue tuning it.

Acknowledgments

Radu Bacioiu, Lin Chen, Xiangyang Liu, and Georgi Vassilev participated in the design and implementation of `DartCVL` while they were at Dartmouth. We are grateful to Rik Faith and Jan Prins of the University of North Carolina at Chapel Hill for supplying us with `UNCVL` source code (which the students refrained from examining while implementing `DartCVL`) and for helpful discussions about the design. Thanks also to Guy Blelloch and Jay Sipelstein of Carnegie Mellon University for technical support. The purchase of cascade was made possible in part by National Science Foundation Grant CDA-9222806.

References

- [BCH⁺93] Guy E. Blelloch, Siddharta Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zaga. *CVL: A C Vector Library Manual, Version 2*. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.
- [Ble92] Guy E. Blelloch. *NESL: A nested data-parallel language*. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992.
- [Dig92] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp Programming Language (ANSI) User's Guide*, December 1992.
- [FHS93] Rickard E. Faith, Doug L. Hoffman, and David G. Stahl. *UnCvL: The University of North Carolina C Vector Library*. Technical Report TR93-063, Department of Computer Science, University of North Carolina at Chapel Hill, May 1993.
- [Pri93] Jan Prins. Private communication, June 1993.