Implementation of a Portable Nested Data-Parallel Language

Guy E. Blelloch Siddhartha Chatterjee[1]
Jonathan C. Hardwick Jay Sipelstein Marco Zagha

February 1993
CMU-CS-93-112

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An earlier version of this paper appeared in "Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming", San Diego, May 1993.

[1]RIACS, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94035.

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the Pittsburgh Supercomputing Center (Grant ASC890018P), who provided Cray Y-MP C90 and Connection Machine CM-2 time, and in part by the National Center for Supercomputing Applications (Grant TRA930102N), who provided Connection Machine CM-5 time.. Guy Blelloch was partially supported by a Finmeccanica chair and an NSF Young Investigator award (Grant CCR-9258525).

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, NASA, Finmeccania, the NSF or the U.S. government.

Keywords: NESL, VCODE, CVL, portable parallel languages, nested parallelism, data parallelism, segmented operations, benchmarks

Abstract

This paper gives an overview of the implementation of NESL, a portable nested data-parallel language. This language and its implementation are the first to fully support nested data structures as well as nested data-parallel function calls. These features allow the concise description of parallel algorithms on irregular data structures, such as sparse matrices and graphs. In addition, they maintain the advantages of data-parallel languages: a simple programming model and portability. The current NESL implementation is based on an intermediate language called VCODE and a library of vector routines called CVL. It runs on the Connection Machines CM-2 and CM-5, the Cray C90, and serial workstations. We compare initial benchmark results of NESL with those of machine-specific code on these machines for three algorithms: least-squares line-fitting, median finding, and a sparse-matrix vector product. These results show that NESL's performance is competitive with that of machine-specific code for regular dense data, and is often superior for irregular data.

1 Introduction

The high cost of rewriting parallel code has resulted in significant effort directed toward developing high-level languages that are efficiently portable among parallel and vector supercomputers. A common approach has been to add data-parallel operations to existing languages, as exemplified by the High Performance Fortran (HPF) effort [33] and various extensions to C (such as C* [49, 47], UC [5], and C** [38]). Such data-parallel extensions offer fine-grained parallelism and a simple programming model, while permitting efficient implementation on SIMD, MIMD, and vector machines. On the other hand, it is generally agreed that although these language extensions are well-suited for computations on dense matrices or regular meshes, they are not as well-suited for algorithms that operate on *irregular structures*, such as unstructured sparse matrices, graphs, or trees [29]. Languages with control-parallel constructs are often better suited for such problems, but unfortunately these constructs do not port well to vector machines, SIMD machines, or MIMD machines with vector processors.

Nested data-parallel languages [10] combine aspects of both data-parallel and control-parallel languages. Nested data-parallel languages provide hierarchical data structures in which elements of an aggregate data structure may themselves be aggregates, and support the parallel application of parallel functions to multiple sets of data. For example, a sparse array can be represented as a sequence of rows, each of which is a subsequence containing the nonzero elements in that row (each subsequence may be a different length). A parallel function that sums the elements of a sequence can be applied in parallel to sum each row of this sparse matrix. Because the calls are to the same parallel function, a technique called *flattening nested parallelism* [17] allows a compiler to convert them into a form that runs efficiently on vector and SIMD machines. Nested data-parallel languages therefore, in theory, maintain the advantages of data-parallel languages (fine-grained parallelism, a simple programming model, and portability) while being well-suited for describing algorithms on irregular data structures. Their efficient implementation, however, has not previously been demonstrated.

As part of the Carnegie Mellon SCAL project, we have completed a first implementation of a nested data-parallel language called NESL. This implementation is based on an intermediate language called VCODE and a library of vector routines called CVL. The implementation runs on the Connection Machines CM-2 and CM-5, the Cray C90, and serial workstations. (We are currently working on a version for workstation clusters.) In this paper we describe the language and its implementation, provide benchmark numbers, and analyze the benchmark results. These results demonstrate that it is possible to get both efficiency and portability on a variety of parallel machines with a nested data-parallel language.

The three benchmarks described in this paper are a least-squares line-fitting algorithm, a median-finding algorithm, and a sparse-matrix vector product. Figure 1 summarizes the benchmark timings. For each machine we give direct comparisons to well-written native code compiled with full optimization. All the NESL benchmark times given in this paper use the interpreted version of our intermediate language (as discussed in Section 5, a compiled version is likely to be significantly faster). The line-fitting benchmark measures the interpretive overhead in our implementation: it contains no nested parallelism and therefore the vectorizing Fortran 77 and CM Fortran compilers generate near-optimal code. The median-finding results show the benefit of NESL's dynamic memory allocation and dynamic load balancing on the Connection Machines. Finally, the sparse-matrix benchmark demonstrates the efficiency of NESL's nested parallel functions on the Cray C90.

The paper is organized as follows: Section 2 describes NESL and illustrates how nested parallelism can be applied to some simple algorithms on sparse matrices. (A description of how NESL can be used for a wide variety of algorithms is given elsewhere [16].) Section 3 outlines the components of the current NESL implementation. Section 4 describes our benchmarks and Section 5 discusses the running times of the benchmarks.

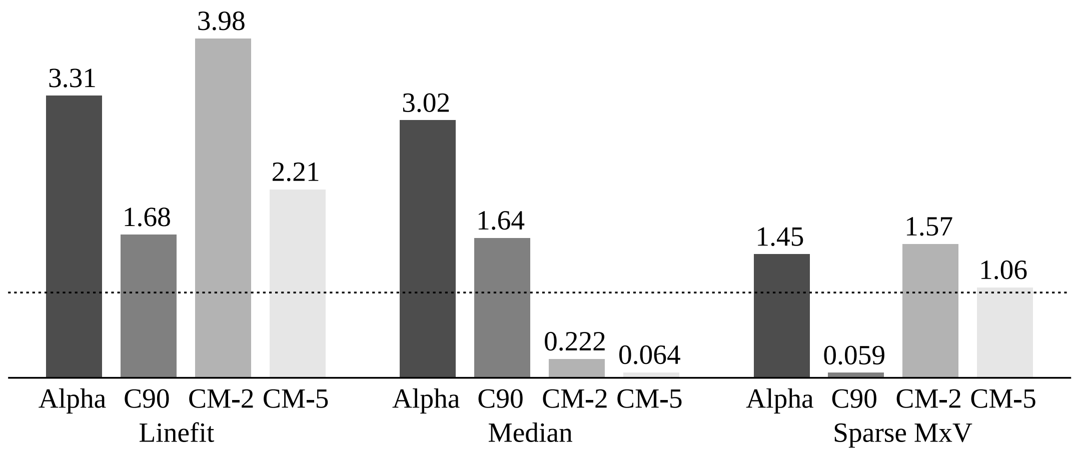

Figure 1: Performance summary for the three benchmarks. The numbers given are the relative performance of NESL and native code versions of the benchmark (smaller numbers are therefore better) on large data sets. Full performance results are given in Section 5.

2 NESL and Nested Parallelism

NESL is a high-level language designed to allow simple and concise descriptions of nested data-parallel programs [11]. It is strongly typed and applicative (free of side-effects). Sequences are the primitive data type and the language provides a large set of built-in sequence operations having efficient parallel implementations. Nested parallelism is achieved through the ability to apply functions in parallel to each element of a sequence. NESL's apply-to-each form is specified using a set-like notation similar to *set-formers* in SETL [52]. For example, the NESL expression

```
{negate(a): a in [3,-4,-9,5] | a < 4}
```

is read as "in parallel, for each `a` in the sequence `[3, -4, -9, 5]` such that `a` is less than `4`, negate `a`". The expression returns `[-3, 4, 9]`. Parallelism is available both in the evaluation of the expression to the left of the colon (`:`) and in the subselection to the right of the pipe (`|`). This parallel subselection can be implemented with packing techniques [6]. NESL also supplies a set of parallel functions that operate on sequences as a whole. Figure 2 lists several of these functions; a full list can be found in the NESL manual [11]. These are similar to operations found in other data-parallel languages.

NESL supports nested parallelism by allowing sequences as elements of a sequence and by permitting the parallel sequence functions to be used in the apply-to-each construct. For example, a sparse matrix can be represented as a sequence of rows, each of which is a sequence of `(column-number, value)` pairs. The matrix

$$m = \begin{bmatrix} 2.0 & -1.0 & \\ -1.0 & 2.0 & -1.0 \\ & -1.0 & 2.0 \end{bmatrix}$$

is represented with this technique as

```
m = [[(0, 2.0), (1, -1.0)],
     [(0, -1.0), (1, 2.0), (2, -1.0)],
     [(1, -1.0), (2, 2.0)]].
```

Operation	Description
`#a`	*Length of sequence* `a`.
`a[i]`	`i`th *element of sequence* `a`.
`sum(a)`	*Sum of sequence* `a`.
`plus_scan(a)`	*Parallel prefix with addition.*
`permute(a,i)`	*Permute elements of sequence* `a` *to positions given in sequence* `i`.
`a -> i`	*Get values from sequence* `a` *based on indices in sequence* `i`.
`a ++ b`	*Append sequences* `a` *and* `b`.

Figure 2: Seven of NESL's sequence functions.

Sum values in each row:
```
  {sum({v : (i,v) in row}): row in m};
```

Delete elements less than `eps`:
```
  {{(i,v) in row | v >= eps}: row in m};
```

Append a column `j` of all 1's:
```
  {[(j,1.0)] ++ row : row in m};
```

Permute the rows to new positions `p`:
```
  permute(m,p);
```

Figure 3: Some operations on sparse matrices. Note that the last operation permutes whole rows.

This representation can be used for matrices with arbitrary patterns of non-zero elements. Figure 3 shows examples of some useful operations on matrices. In these operations there is parallelism both within each row and across the rows. The available parallelism is therefore proportional to the total number of non-zero elements, rather than the number of rows (outer parallelism) or average row size (inner parallelism). Graphs can be represented analogously, using subsequences to store adjacency lists.

Nested parallelism is also very useful in divide-and-conquer algorithms. As an example, consider a parallel quicksort algorithm (Figure 4). The three assignments for `les`, `eql` and `grt` select the elements less than, equal to, and greater than the pivot, respectively. The expression

```
  {qsort(v):v in [les, grt]}
```

puts the sequences `les` and `grt` together into a nested sequence and applies `qsort` in parallel to the two elements of this sequence. The result is a sequence with two sorted subsequences. The concatenation function `++` is then used to append the three sequences. In this algorithm, there is parallelism both within the selection of each of the three intermediate sequences and in the nested parallel execution of the recursive calls. A flat data-parallel language would not permit the recursive calls to be made in parallel.

We decided to define a new language instead of adding nested parallel constructs to an existing language for two main reasons. First, we wanted a small core language, to allow us to guarantee that everything that is expressed in parallel compiles into a parallel form. Second, we wanted a side-effect-free language because implementing (and defining consistent semantics for) nested data-parallelism when nested function calls can interact with each other through side-effect is very difficult.

Although we feel that it would be possible to add nested data-parallelism to an imperative language, we

```
function qsort(s) =
if (#s < 2) then s
else
  let pivot = s[rand(#s)];
      les = {e in s| e < pivot};
      eql = {e in s| e = pivot};
      grt = {e in s| e > pivot};
      result = {qsort(v):v in [les, grt]}
  in result[0] ++ eql ++ result[1];
```

Figure 4: A nested data-parallel quicksort in NESL.

doubt that nested data-parallelism could be added to C or Fortran in such a way that the resulting language would be both clean and efficient. For C it is likely that one would have to limit the type of objects permitted in the parallel data structure in order to produce efficient code. In particular, allowing arbitrary pointers in a parallel structure, while highly desirable, would also be very hard to implement. For Fortran 77, the static memory requirements would severely limit the usefulness of a nested data-parallel language. The additional data management features in Fortran 90 could reduce these limitations, but the many features of the language are so delicately balanced that permitting nested structures would likely topple it.

3 System Overview

The full implementation of NESL consists of a NESL compiler, an intermediate language called VCODE [12], an interpreter for VCODE, and a portable library of parallel routines called CVL [13]. We also have an experimental VCODE compiler for shared-memory MIMD machines [19, 20]. The roles of the different components are shown in Figure 5. This section gives an overview of the each of these components.

The NESL execution times reported in this paper are for interpreted VCODE. Use of an interpreted intermediate language allows us to port our system very quickly to new machines; the only component that needs to be rewritten is a library of vector routines called by the interpreter. There is an efficiency loss from using an interpreter; this loss depends on the particular machine and on the problem size. One portion of the overhead is a fixed cost per executed VCODE instruction (a single VCODE instruction operates on an entire sequence). This constant overhead is amortized over the per-element cost of executing a sequence operation, so the system attains higher efficiency if longer sequences are used. Our technique for compiling nested parallelism (flattening) allows us to achieve the efficiency of operating on single long sequences instead of several shorter sequences in the nested parallel calls. These efficiency tradeoffs are analyzed quantitatively in Section 5.

3.1 NESL compiler

The NESL compiler translates NESL code into VCODE. It is written in Common Lisp [55] and is invoked within an interactive environment. Figure 6 shows the phases of the compiler. This section concentrates on the third phase, *flattening nested parallelism*, and briefly discusses the fourth phase, *type specialization*. The type checker in the second phase is a Milner style type checker [42] with type inference, and the other phases use standard compiler techniques.

Our approach to implementing nested parallelism uses a technique called flattening nested parallelism [17, 10, 46]. This process converts nested sequences into sets of flat vectors and translates the

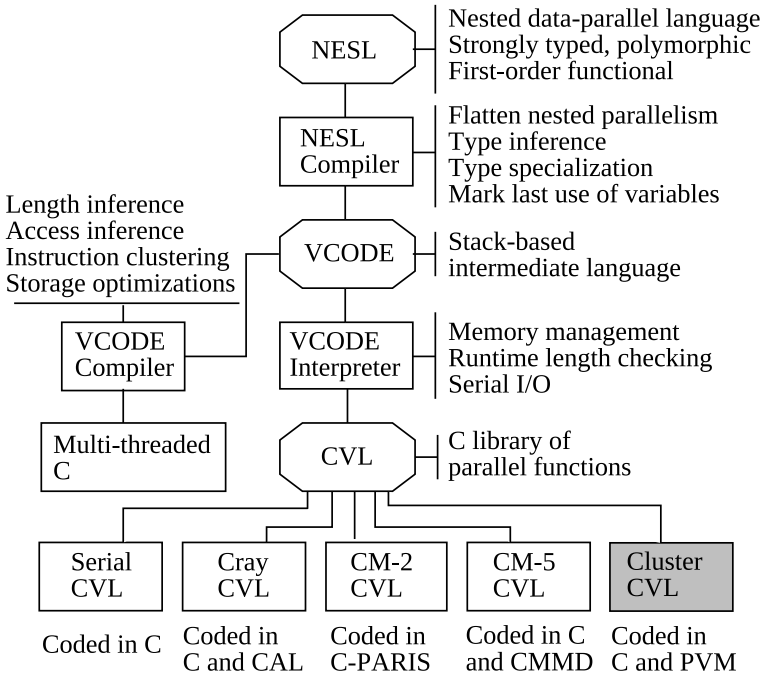

Figure 5: The parts of the SCAL project and how they fit together. CAL stands for Cray Assembly Language, C-PARIS is the C interface to the CM-2 Parallel Instruction Set, and CMMD is the CM-5 message-passing library. Cluster CVL is under development.

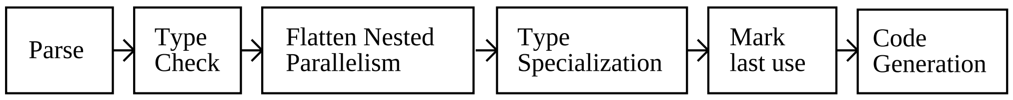

Figure 6: The main tasks of the NESL compiler.

nested operations into segmented[1] VCODE operations on the flattened representation. The flattening of nested sequences is achieved by converting each sequence into a pair: a `value` vector and a set of segment descriptors. For example, the sequence `[2, 9, 1, 5, 6, 3, 8]` would be represented by the pair

```
segdes = [7]
value  = [2, 9, 1, 5, 6, 3, 8]
```

and the nested sequence `[[2, 1], [7, 0, 3], [4]]` would be represented as

```
segdes1 = [3]
segdes2 = [2, 3, 1]
value   = [2, 1, 7, 0, 3, 4]
```

In these examples, a `segdes` with only one value specifies that the whole vector is one segment (the use of this seemingly redundant data is critical when implementing nested versions of user-defined functions).

[1]Segments are a technique for implementing nested parallelism and are fully described by Blelloch [8, 9, 10].

In the second example, `segdes1` describes the segmentation of `segdes2`, not of `value`. Sequences that are nested deeper are represented by additional segment descriptors. Sequences of fixed-sized structures (such as pairs) are represented by multiple `value` vectors (one for each slot in the structure) that share a common segment descriptor.

Using this representation, nested versions of NESL operations with VCODE counterparts can be directly converted into the corresponding segmented VCODE instruction. For example, VCODE includes a segmented `+-reduce` operation, which if passed the two vectors shown above, would return the vector `[3, 10, 4]`. Nested versions of user-defined functions are implemented by using program transformations called *stepping-up* and *stepping-down* [17]. These transformations convert all the substeps of a nested call into segmented operations or into calls to other functions that have already been transformed. With these transformations, when a user function is used in an apply-to-each form, the data for each subcall is in a separate segment, and computations on each segment can proceed independently.

The most complex part of flattening nested parallelism is transforming conditional statements. There are two main parts of this transformation. The first part inserts code for packing out all the data in subcalls that do not take a branch and for reinserting this data when the branch is complete. This guarantees that work is only done on the subcalls that take the branch, and is similar to techniques used by vectorizing compilers to vectorize loops with conditionals [64, §3.6]. It also results in proper load balancing on parallel machines. The second part inserts code to test if any of the subcalls are taking the branch and to skip the branch if none are. This is important for guaranteeing termination. The communication costs involved in doing the packing and unpacking can be quite high, and one area of our ongoing research is to see how the communication can be reduced by only packing when there is a significant load imbalance among processors [53].

We now briefly describe the third phase of the NESL compiler, which involves specializing polymorphic code. The standard implementation of polymorphism is to have a single compiled version of a function and achieve polymorphism by use of pointers—the function only manipulates pointers to the data rather than the actual data and therefore doesn't care about the type or size of the data. In this scheme there is almost no control of layout of the data since such control would require knowledge about the size of the data, which depends on its type. Because of the high cost of communication among processors on parallel computers, it is important that we have control over the layout of data. The NESL compiler therefore specializes all polymorphic functions to specific types and generates code for each type. The type system is such that the compiler can determine all types to which a particular function is going to be applied at compile time. The compiler must, however, have access to the whole program.

3.2 VCODE

VCODE [12, 14] was designed as a testbed for a systematic study of compiler and implementation issues that arise in data-parallel languages. Accordingly, its design concentrates on data parallelism to the exclusion of other issues more commonly seen in language designs, such as data structures and advanced control constructs.

The aggregate data type in VCODE is the homogeneous segmented vector of atomic types. There are four basic data types: boolean vectors, integer vectors, floating-point vectors, and segment descriptors. Segment descriptors describe how vectors are partitioned into segments. For example, the segment descriptor `[2, 3, 1]` specifies that a vector of length 6 should be considered as 3 segments of lengths 2, 3 and 1, respectively (segments are contiguous and non-overlapping). Each instruction then operates independently over each segment of the vector. For example, the `+_scan` instruction performs a parallel prefix operation on each segment, starting from zero on each segment boundary. The segmented versions of the instructions are critical for the implementation of nested parallelism (see Section 3.1). The notion of segments also appears in some of the library routines of the Connection Machines CM-2 [57] and CM-5 [60] and has been adopted in the PREFIX and SUFFIX operations of High Performance Fortran [33]. The internal VCODE representation of the segment descriptor is machine-dependent: our serial implementation uses a sequence of the lengths of each segment, while our implementations on the Cray and the Connection Machines CM-2 and CM-5 also use flags to mark boundaries between segments.

VCODE can dynamically create and destroy arbitrarily long vectors, as this is necessary for most data-parallel languages. Unlike machine-oriented languages, VCODE provides no notion of processors, either physical or virtual. This allows the interpreter or compiler to choose strategies for process mapping and memory management appropriate for the target machine.

VCODE is based on a stack model of memory and resembles Pcode [44], a serial intermediate language designed for Pascal. It is conveniently defined in terms of an abstract machine, the vector stack machine. This is a standard stack machine, except that each stack location contains an arbitrarily long vector of atomic values. Each vector has a length associated with it, and vectors in different positions on the stack may have different lengths. Instructions operate on entire vectors at a time. For example, the + instruction pops the top two vectors (of equal length and appropriate type) from the stack, adds corresponding elements together, and returns the result vector to the top of the stack.

VCODE contains a small set of instructions, most of which operate on vectors of atomic values. The vector instructions operate on a fixed number of vectors from the top of the stack and return their result to the top of the stack. The vector instructions can be divided into the following subclasses: *elementwise instructions*, which are apply-to-each extensions of arithmetic and logical operations; *scan and reduction instructions*, which provide reduction and parallel prefix operations on vectors for a fixed set of associative operators; *permute instructions*, which rearrange elements of a vector according to another vector of indices, possibly accompanied by masking or default vectors; *segment descriptor manipulation instructions*, which create and manipulate segment descriptors; and *I/O instructions*, which read and write vectors and segment descriptors. The permute instructions can be used to implement vector-based indexing in C*, Fortran 90, and APL, for the β operation in CM-Lisp [63], and for the `move` operation in Paralation Lisp [51]. The segmented instructions permit the implementation of the nested parallelism allowed in CM-Lisp, SETL, and Paralation Lisp. The rest of the VCODE instructions are for flow control, stack management and system interface. Of special note are the stack management functions `POP` and `COPY`, each of which takes two arguments. These specify a distance up the stack to start and a number of elements on which to operate.

A VCODE program is a set of VCODE functions. VCODE functions are maps from stack prefixes to stack prefixes; the stack is used to pass parameters and return results. Control flow within a function is directed by a conditional form and recursion. The conditional form has the restriction that both branches must leave the stack in the same state with respect to depth and data types. This is necessary to ensure that functions are well-formed. All instructions are strict, i.e., all inputs must be evaluated before an instruction can be executed. Synchronization is implicit and lock-step: an instruction must be completed before the next instruction can begin execution. VCODE is side-effect free, so some permutation instructions may require the destination to be copied before any data movement occurs.

3.3 VCODE interpreter

The two main requirements for the VCODE interpreter are portability and efficient management of vector memory. The interpreter is written in ANSI C to ease portability and contains little machine-dependent code (most of which stems from operating system differences). The typical sequence of operations performed by the interpreter to execute a vector operation is as follows: a table lookup is performed to find the number of stack arguments an operation uses and the length and type of the operation's result; optionally, a check is made that any length and type constraints on the arguments are satisfied; memory for the result is allocated, as explained below; the associated CVL function is called to perform the vector operation; after completion,

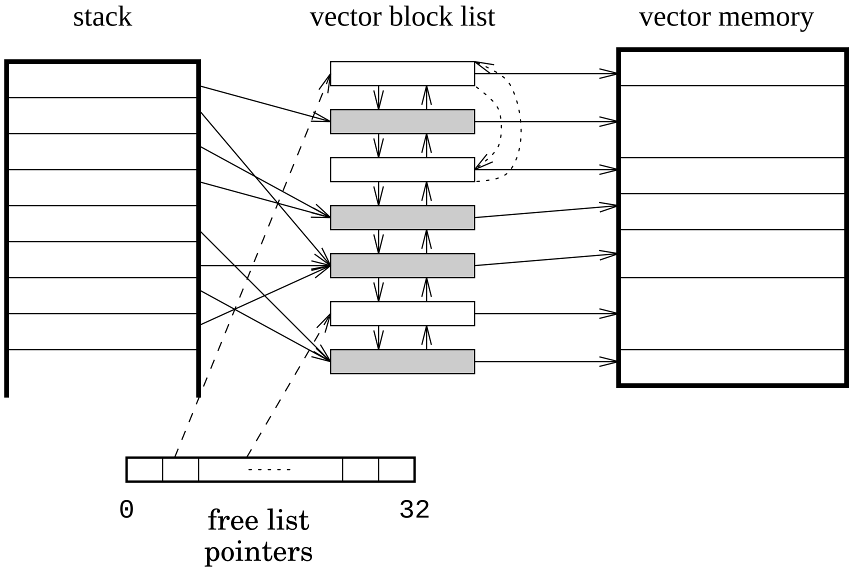

Figure 7: Internal structure of the VCODE interpreter. The dark entries in the vector block list refer to active regions of vector memory. The light ones correspond to free regions. Entries to free regions of similar size are linked together in the same free list structure. There are 32 free lists, each one referring to regions at most twice the size of the previous.

the arguments are popped off the stack. All this results in a constant time interpretive overhead per executed VCODE instruction.

The most novel aspect of the interpreter is its memory management. VCODE creates and destroys vectors of differing and dynamically determined sizes. Most memory management and garbage collection literature (see [3], for example) assumes large numbers of small objects and a large (virtual) memory. VCODE programs do not behave in this manner. In particular, there are usually few large data structures (vectors) active at any one time, and we want to operate on the largest possible problem size that available memory can support. The supercomputers on which VCODE runs typically provide no virtual memory facilities, so there is a hard limit on the amount of memory a program may use. Therefore we must be very thrifty with the amount of memory used by a VCODE program.

Figure 7 shows the internal data structures for memory management. Each entry in the vector stack is really a pointer to an entry in a *vector block list*. VCODE stack modification operations are implemented via pointer manipulations; no action is taken on vector memory.

Entries in the vector block list describe how vector memory is currently partitioned. The vector block list maintains an order-preserving 1-1 map to regions of vector memory (in terms of Figure 7, the arrows from the list to vector memory never cross). Each region of vector memory is either active or free, as indicated by a flag in the corresponding vector block entry. Each vector block entry has a reference count, giving the number of times it occurs on the stack. When this number reaches zero, the vector stored in the region can no longer be referenced. The interpreter frees the associated region of memory, merges it with any adjacent free regions (destroying the vector block entry of the merged region) and puts it on a *free-list*. There are several free-lists, each corresponding to a different range of region sizes (currently, each list has regions a factor of two larger than the previous list). Use of multiple free lists for different sized regions allows us to examine fewer regions to find one of the appropriate size.

To allocate space for a new vector of known memory size, the interpreter tries to find a large enough free region of vector memory by doing a first-fit on the appropriate free-list. If no block in that free-list is big enough, the free-lists of larger region sizes are checked in a first-fit manner, and if none is found, the *garbage compacting* routine is called. This routine pushes all vectors as far as possible to the front of vector memory, creating one large region of free memory at the end. If this region is not large enough, the interpreter signals an out-of-memory error. When a large enough region has been found, the interpreter divides it into two pieces: one for the vector and one that is returned to the free-list appropriate for that region's size.

The reference counts in the vector block list also enable the interpreter to perform simple copy elimination optimizations [28, 31]. To enforce the applicative semantics of VCODE the implementation of an operation that changes only a single element of a vector must copy the entire vector before making any alteration. If the reference count indicates that this is the last reference to the "old" version of the vector, we may avoid the copy and implement the operation in a (more efficient) destructive manner.

3.4 VCODE compiler

Chatterjee's doctoral dissertation [19, 20] discusses the design, implementation, and evaluation of an optimizing VCODE compiler for shared-memory MIMD machines. In this section, we give a summary of these issues.

There is, of course, a trivial implementation of VCODE on a MIMD machine: each VCODE instruction is written as a parallel loop, and the processors synchronize between instructions. This is precisely how the VCODE interpreter works. However, such an implementation of data parallelism on a MIMD machine has several performance bottlenecks, which limit both the asymptotic performance and the performance for small problem sizes.

- The parallelism of the source language is too fine-grained for MIMD machines. Since loop overhead scales with problem size, it limits the asymptotic performance of the parallel program.
- Serial overhead related to load balancing, storage management and loop setup is independent of the problem size, and therefore influences the small problem size performance but not the asymptotic performance.
- The implicit lock-step synchronization of the data-parallel model is expensive to implement on MIMD machines, whereas it comes for free on SIMD machines. Since the cost of barrier synchronization on a MIMD machine depends on the number of processors but is independent of problem size, this only affects the performance at small problem sizes.
- The (potentially large) intermediate results generated by the fine-grained parallelism cause problems for the memory organizations typically found in MIMD machines. In particular, locality of reference and its concomitant benefits are compromised. Loss of locality increases data access times; it scales with the problem size, and limits asymptotic performance.

The VCODE compiler addresses these problems through a set of program optimization techniques that cluster computation into larger loop bodies, reduce intermediate storage, and relax synchronization constraints. These techniques do not require knowledge of data sizes or the number of processors at compile time. There are four principal optimization techniques:

- *Size inference* symbolically identifies equivalence classes of data sizes and loop structures.
- *Clustering* uses the information produced by size inference and uses it to partition the computation graph into a set of parallel loops that can be scheduled.
- *Access inference* refines the loops produced by clustering based on conflicts between definition and usage access patterns of data vectors, producing loops free of any explicit synchronization.

- *Storage optimizations* remove array storage or reduce the storage from a full array to a smaller buffer storing only a few elements.

The compiler accepts VCODE as input and outputs C code with thread extensions targeted at shared-memory multiprocessors. The native C compiler is then invoked to produce the final machine code. This makes the compiler portable and allows the use of the best compiler technology available for the machine.

3.5 CVL

To enable rapid porting of VCODE to new machines, we designed CVL (C Vector Library) [13], a library of low-level segmented vector routines callable from C. These are used by a VCODE interpreter, described in Section 3.3.

The purpose of CVL is to provide a portable segmented vector abstraction that can be efficiently implemented on a wide range of machines. This library can then be used by interpreters and compilers of more complicated languages. Examples of the vector routines supplied by the library include: numerous elementwise operations (arithmetical and logical), scan and reduction operations, and a variety of permutation operations. All primitives are defined for both unsegmented and segmented vectors, and for several data types: integers, booleans, and double precision floating point numbers. Although CVL implementations are machine-specific, the interface to the library is machine-independent. Thus C code written using CVL is portable across all machines with a C compiler.

The vector memory model supplied by CVL was designed to be portable to both shared and distributed memory machines. CVL stores vectors in a region of memory (*vector memory*) that should be viewed as separate from normal memory. CVL exports only a handle to a vector, known as a `vec_p`. Non-library functions may not do anything with a `vec_p` except store it and pass it as a parameter to CVL routines. Given a `vec_p`, there are CVL functions that return the amount of vector memory that the corresponding vector takes. CVL itself does no memory management, other than the initial allocation of the totality of the vector memory. It is the responsibility of the code using the library to manage this memory. One way of doing this is described in Section 3.3.

The key to efficient implementation of nested data-parallelism is the efficient implementation of operations on segmented vectors, which in our system corresponds to efficient code for the various CVL functions. Unfortunately current supercomputer compilers cannot adequately compile Fortran or C implementations of these operations, forcing us to implement these operations at a very low-level for the target machines.

3.5.1 Cray CVL

We have implemented Cray CVL for a single processor of the Cray Y-MP and Y-MP C90. The CVL library consists of Cray Assembly Language [24] kernels which are called from C.

The choice of Cray Assembly Language (CAL) might seem unnecessarily low-level, especially for implementing a vector language on a vector machine. However, while the elementwise operations could be implemented efficiently in C or Fortran, it is very difficult to produce efficient code for the scan operations and segmented operations from C or Fortran. There are several reasons why hand-written assembly code outperforms C or Fortran code. For segmented operations, the segmentation of a vector is represented using *packed booleans,* a representation that is not supported by high-level languages. Storing 64 booleans in a single word reduces memory requirements and makes efficient use of the Cray's functional units. The scan operations also need to use a non-standard loop structure we call *loop raking* [15] in order to be vectorized. Finally, in order to make the best use of hardware chaining and avoid register reservation effects [48], it is necessary to unroll the loops. The scan and segmented scan algorithms are described in detail elsewhere [21].

CAL code is very efficient, but programming in raw CAL is tedious, error-prone, and non-portable. In order to program at a higher-level, we implemented the CVL kernels using our own macro assembler (written in Common Lisp) that generates CAL. In the assembler, we developed a set of loop macros for implementing scans, including a version that generates unrolled loops. Part of the motivation for developing the macro assembler was to allow us to port the code easily to other vector machines in the future.

Although the current version of Cray CVL only uses one processor, we have implemented prototypes of multiprocessor vector operations [15] and we are working on a complete multiprocessor library using the Cray microtasking facilities [25].

3.5.2 CM-2 CVL

The implementation of CVL for the CM-2 is built on top of the Paris instruction set [57]. As a machine-specific language, Paris has several nice features for implementing CVL: a clean interface to C, a model of the CM-2 as a collection of virtual processors, and direct support for scan, reduce, and other vector operations. It also has three significant drawbacks, however. First, it uses the older "fieldwise" representation of data, which involves extra transpose operations when using the floating point chips. The newer and more efficient "slicewise" representation is not supported by Paris. Second, Paris vectors (the basic Paris data type) are limited to 64K bits in size, and the current CVL implementation inherits this limitation. Third, assigning the responsibility of memory management to VCODE means that CVL vectors are really pointers into a single big block of memory. Since Paris communication functions cannot operate on these pointers (implemented as Paris "aliases" with non-zero offsets), CVL arguments must be copied into temporary vectors before use.

This problem and its inelegant solution illustrate the difficulty of finding an intermediate language model that can be efficiently implemented across all supercomputer architectures. In this case (CM-2 Paris), it would be more efficient if the job of memory allocation were moved from VCODE to CVL. This would let the CVL code allocate new Paris vectors on demand, thereby eliminating the overhead of double copying. Alternatively, CM-2 CVL could be rewritten at a much lower level in PEAK [50] code, which would bypass all of the Paris-based limitations, but at a high cost in terms of programmer effort and maintainability.

3.5.3 CM-5 CVL

For the CM-5 implementation of CVL, we originally thought that CM Fortran [59] would be the language of choice. This has the attraction of being available on the CM-2 as well, and of generating slicewise PEAK code for that machine. Thus an implementation of CVL written in CM Fortran for the CM-5 should be easily portable back to the CM-2 and holds the promise of offering improved performance over the existing Paris-based CM-2 CVL. Furthermore, CM Fortran code can use the CM-5 vector units, which greatly increase the speed of simple elementwise operations. However, CM Fortran does not provide low-level access to the underlying machine. This caused several major problems. There was no way to force the compiler to layout CVL memory efficiently (since the compiler could not know ahead of time how the memory would be subdivided by VCODE). Similarly, there was no way to guarantee maximal alignment of CVL vectors, which the VCODE interpreter relies upon to be able to, for example, place a vector of doubles where a vector of booleans used to start.

We therefore wrote CM-5 CVL in C, using the CMMD [60] message-passing library. This combination gives full control over memory layout and alignment, allowing vectors to be load-balanced across all processors. The CVL scan and reduction functions are implemented using the corresponding CMMD primitives, while functions involving arbitrary communication (such as permute and pack) use the CMAML Active Message functions [62]. Our current implementation is an initial release that does not fully reflect the power of the machine. Future improvements are expected to include:

- Using CMNA [61] to access the network interface chips directly. With specialized receive functions, this will allow the full payload of each packet to be used, rather than sacrificing one word to the active message system. Direct access to the on-chip packet counters will also reduce the time taken to agree on whether all packets have been sent and received.
- Using CDPEAC [58] to exploit the vector units. This will greatly increase the speed of elementwise operations.

3.5.4 Other CVL implementations

Our implementation of CVL on workstations uses ANSI C and can therefore be ported easily to most serial machines. Cluster CVL is being written using C and PVM [30]. Faith *et al.* [27] at the University of North Carolina have ported CVL to the Maspar MP-2 using the MPL programming language.

4 Benchmarks

This section describes three benchmarks: a least-squares line-fit, a generalized median find, and a sparse-matrix vector product. The particular benchmarks were selected for their diverse computational requirements, summarized in Figure 8. They are each simple enough that the reader should be able to fully understand what the algorithm is doing, but are more complex than bare kernels such as the Livermore Loops [41]. The performance of these benchmarks demonstrates many of the advantages and disadvantages of our system. The results of these benchmarks will be analyzed in Section 5.

	Communication	Dynamic Structures	Nested Parallelism
Line Fit	Low	No	No
Median	High	Yes	No
Sparse MxV	High	No	Yes

Figure 8: The properties of the benchmarks.

Our first benchmark is a least-squares line-fitting routine using the algorithm described in Press *et al.* [45, §14.2]. The version we use assumes all points have equal measurement errors. This is a straightforward algorithm that requires very little communication and has no nested parallelism. Furthermore, all vectors are of known size at function invocation and can be allocated statically. It is therefore well-suited for languages such as Fortran 90. We use this benchmark to measure the overhead incurred by our interpreter-based implementation. The NESL code for the benchmark is given in Figure 9.

Our second benchmark is a median-finding algorithm. To implement median-finding we use a more general algorithm that finds the k^{th} smallest element in a set. The algorithm is based on the quickselect method [34]. This method is similar to quicksort, but calls itself recursively only on the partition containing the result (the recursion was removed in the Fortran version). This algorithm requires dynamic memory allocation (also removed in the Fortran version) since the sizes of the less-than-pivot and greater-than-pivot sets are data dependent. In order to obtain proper load balancing, the data must be redistributed on each iteration. The NESL code for the algorithm is shown in Figure 10. This algorithm was selected to demonstrate the utility and efficiency of NESL's dynamic allocation.

Our third benchmark multiplies a sparse matrix by a dense vector and demonstrates the power and efficiency of nested parallelism. Sparse-matrix vector multiplication is an important supercomputer kernel

```
function linefit(x, y) =
let
  n    = float(#x);
  xa   = sum(x)/n;
  ya   = sum(y)/n;
  Stt  = sum({(x - xa)^2:  x});
  b    = sum({(x - xa)*y:  x; y})/Stt;
  a    = ya - xa*b;
  chi2 = sum({(y-a-b*x)^2:  x; y});
  siga = sqrt((1.0/n + xa^2/Stt)*chi2/n);
  sigb = sqrt((1.0/Stt)*chi2/n)
in
  (a, b, siga, sigb);
```

Figure 9: NESL code for fitting a line using a least-square fit. The function takes sequences of `x` and `y` coordinates and returns the intercept (a) and slope (b) and their respective probable uncertainties (`siga` and `sigb`).

that is difficult to vectorize and parallelize efficiently because of its irregular data structures and high communication requirements. While there are many algorithms for special classes of sparse matrices, we are interested in supporting operations for arbitrary sparse matrices. This is challenging since the matrices used in a number of different scientific and engineering disciplines often have average row lengths of less than 10. These row lengths are significantly less than the start-up overhead for vector machines ($n_{1/2}$) and are far too small to divide among processors in an attempt to parallelize row by row. On the other hand, dividing rows among processors makes load balancing difficult since each row can have a different length and the longest rows could be very much longer than the shortest. Our implementations (in NESL, C, and Fortran) use a compressed row format containing the number of non-zero elements in each row, and the values of each non-zero matrix element along with its column index [26]. Figure 11 shows the NESL implementation and Figure 12 shows an example of a sparse-matrix vector product using this format.

5 Results

Running times for all benchmarks with a variety of data sizes are given in Table 1. Times are given for both interpreted NESL code and for native code. For native code we used Fortran 77 on the Cray C90, CM Fortran [59] on the Connection Machines CM-2 and CM-5, and C on the DEC Alpha workstation. In all cases we used full optimization. The native code we used is given in Appendix A. NESL timings are for the code shown in Section 4, run using the VCODE interpreter. All Alpha benchmarks were run on a DEC 3000 AXP Model 400 with 32 Mbytes of memory. All Cray C90 benchmarks were run on one processor of a C90/16 with 256 Mwords of memory. All Connection Machine CM-2 benchmarks were run on 32K processors of a CM-2, with 32 Kbytes of memory per processor. All Connection Machine CM-5 benchmarks were run on 256 processors of a CM-5, with 32 Mbytes of memory per processor.

The CM-5 CM Fortran benchmarks did not use the vector units, in order to enable a better comparison to be made with the current implementation of CVL on that machine. When the vector units are used, the CM Fortran line-fit benchmark runs 1–40 times faster (depending on problem size), the CM Fortran median benchmark runs 2–5 times faster, and the CM Fortran sparse-matrix vector product benchmark runs 1–1.5

```
function select_kth(s, k) =
let pivot = s[#s/2];
    les = {e in s | e < pivot}
in
  if (k < #les) then
    select_kth(les, k)
  else
    let grt = {e in s | e > pivot}
    in if (k >= #s - #grt) then
         select_kth(grt, k - (#s - #grt))
       else pivot;

function median(s) = select_kth(s, #s/2);
```

Figure 10: NESL code for median finding. The function `select_kth` returns the `k`th smallest element of `s`. This is used by `median` to find the middle element.

```
function MxV(Mval, Midx, Mlen, Vect) =
let v = Vect -> Midx;
    p = {a * b:  a in Mval; b in v}
in
  {sum(row) :  row in nest(p, Mlen)};
```

Figure 11: NESL code for sparse-matrix vector product. `Mval` holds the matrix values, `Midx` holds the column indices, `Mlen` holds the length of each row, and `Vect` is the input vector. The function `nest` takes the flat sequence `p` and nests it using the lengths in `Mlen` (the sum of the values in `Mlen` must equal the length of `p`).

$$\begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 4 & 0 & 0 & 2 \\ 3 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 10 \\ 20 \\ 30 \\ 40 \end{pmatrix} = \begin{pmatrix} 30 \\ 60 \\ 120 \\ 50 \end{pmatrix}$$

`Vect` =	`[10 20 30 40]`
`Midx` =	`[ 0  2  0  3  0  1]`
`Vect->Midx` =	`[10 30 10 40 10 20]`
`Mval` =	`[ 3  2  4  2  3  1]`
`p` =	`[30 60 40 80 30 20]`
`Mlen` =	`[ 1  1  2  2]`
`nest(p, Mlen)` =	`[[30] [60] [40 80] [30 20]]`
`rowsums` =	`[30 60 120 50]`

Figure 12: An example of sparse-matrix vector product.

n	Alpha C	Alpha NESL	C90 F77	C90 NESL	CM-2 CMF	CM-2 NESL	CM-5 CMF	CM-5 NESL
				Line Fit				
2^{10}	0.0007	0.0029	0.0001	0.0012	0.0018	0.0061	0.0008	0.0039
2^{14}	0.0137	0.0468	0.0004	0.0018	0.0019	0.0061	0.0011	0.0040
2^{18}	0.2869	0.9506	0.0058	0.0122	0.0037	0.0133	0.0057	0.0101
2^{22}			0.0927	0.1551	0.0322	0.1283	0.1473	0.3251
				Median				
2^{10}	0.0004	0.0059	0.0001	0.0059	0.0293	0.1017	0.0086	0.0185
2^{14}	0.0068	0.0273	0.0005	0.0092	0.0623	0.1442	0.0215	0.0285
2^{18}	0.1347	0.4070	0.0080	0.0233	0.2667	0.2163	0.2146	0.0596
2^{22}			0.1276	0.2099	3.7810	0.8389	8.2092	0.5243
				Sparse-Matrix Vector Product				
2^{10}	0.0002	0.0009	0.0002	0.0003	0.0043	0.0142	0.0012	0.0010
2^{14}	0.0049	0.0088	0.0037	0.0006	0.0063	0.0152	0.0020	0.0035
2^{18}	0.1503	0.2186	0.0589	0.0038	0.0295	0.0451	0.0175	0.0256
2^{22}			0.9436	0.0557	0.4098	0.6436	0.2791	0.2960

Table 1: Running times (in seconds) of the benchmarks for NESL and native code. The sparse-matrix vector product uses a row length of 5 and randomly-selected column indices. CM-5 NESL results are preliminary.

times faster. It is expected that a future version of CM-5 CVL will exploit the vector units and be able to achieve similar speedups.

We now discuss three main issues exhibited by the timings: the advantage of nested parallelism in the implementation of the sparse-matrix vector product, the overhead incurred by our interpreter, and the need for dynamic load-balancing in the median-finding code on the Connection Machine CM-2.

Nested Parallelism: The sparse-matrix vector product benchmark demonstrates the advantages and efficiency of nested data parallelism. Figure 13 gives running times on the Cray for a variety of degrees of sparsity. For very sparse matrices, the NESL version outperforms the native version by over a factor of ten. This performance gain arises because the compilation of nested data parallelism described in Section 3.1 generates code with running time essentially independent of the size of the sub-sequences. The nested code achieves full efficiency (vectorization on the Cray and high data-to-processor ratio on the CM-2 and CM-5) by executing on the full input data. The result is consistently high performance regardless of the sparsity of the matrix. Note, however, that as the matrix density increases, the Cray Fortran performance improves. Eventually, Fortran achieves superior performance because of NESL's extra per-element cost of interpretation relative to compilation.

Interpretive overhead: The main source of inefficiency in our system is the interpretation of the VCODE generated by the NESL compiler. The cost of interpretation can be analyzed by studying the line-fitting benchmark since this benchmark requires very little communication and the native-code implementations compile to almost perfect code.

There are two main sources of interpretive overhead in our system. First, there is the cost of executing

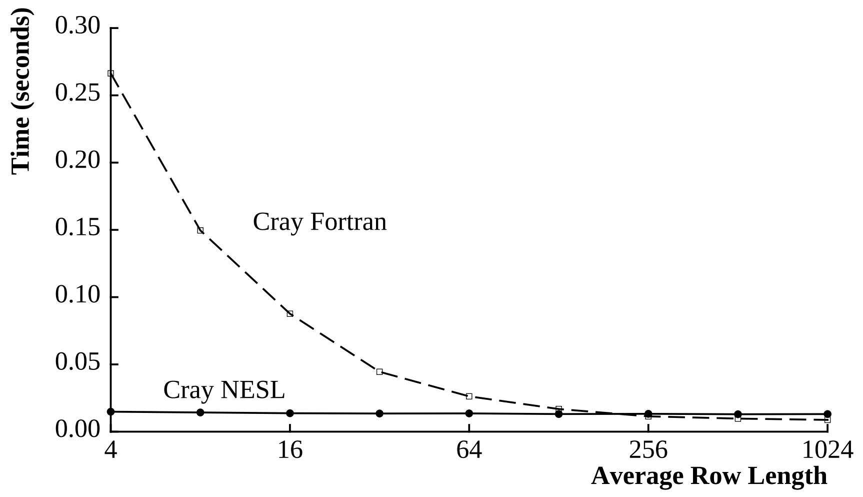

Figure 13: Running times of the sparse-matrix vector product for varying levels of sparsity. The number of nonzero entries in each sparse matrix is fixed at 10^6.

the interpreter itself. For the line-fitting benchmark, this is constant, independent of input size (since the interpreter executes a fixed number of VCODE steps), and so it can be computed by examining the running times for small input. Figure 14 shows the percentage of run time accounted for by this overhead for varying input sizes, as well as the $n_{1/2}$ value at which the implementations attain half of their asymptotic efficiency. As the figure shows, NESL sometimes requires fairly large input in order to attain close to its peak efficiency. This overhead is not a problem on the CM-2; since there are 32K processors, the loss of efficiency when working with small vectors ($n <$ 32K) overwhelms the interpretive overhead.

The second major deficiency of an interpreter-based system is that the granularity of the operations performed by the library is too fine. Each operation on a collection of data is performed by a distinct call to the CVL library. In a compiled system, the loops performing the separate parallel operations could be fused together. This optimization would result in much better memory locality (quantities could be kept in registers and reused, instead of being loaded from memory, acted on, and written back) and would also allow chaining on the Cray. These loop fusion operations are performed by the VCODE compiler [19, 20]. With the interpreter these inefficiencies adversely affect the peak performance of NESL programs, and their effects can be seen in the performance of line-fitting for large data sizes (see Table 1). On the CM-2 there is an additional important source of inefficiency: CM-2 CVL is built on top of the Paris instruction set [57]. Although working with Paris has many advantages, it forces use of the older “fieldwise” representation of data, instead of the more efficient “slicewise” representation generated by the CM Fortran compiler.

Dynamic load balancing: We now consider why the native code for the median algorithm does poorly compared to the NESL code on the CM-2. The median algorithm reduces the number of active elements on each step. In our CM Fortran implementation, as these elements get packed to the bottom of an array, they become more imbalanced across the processors. Although it is possible to pack the elements into a smaller array, this would require dynamically allocating a new vector on each step, which is awkward in CM FortranİIn NESL, vectors are dynamically allocated with the data automatically balanced across the processors. The NESL implementation of the median algorithm only requires a total of $O(n)$ work because on each of the $O(\log n)$ steps the amount of data processed is cut by a constant factor. Since the CM Fortran implementation requires $O(n)$ work on each step, it is a factor of $O(\log n)$ slower, as illustrated in Figure 15.

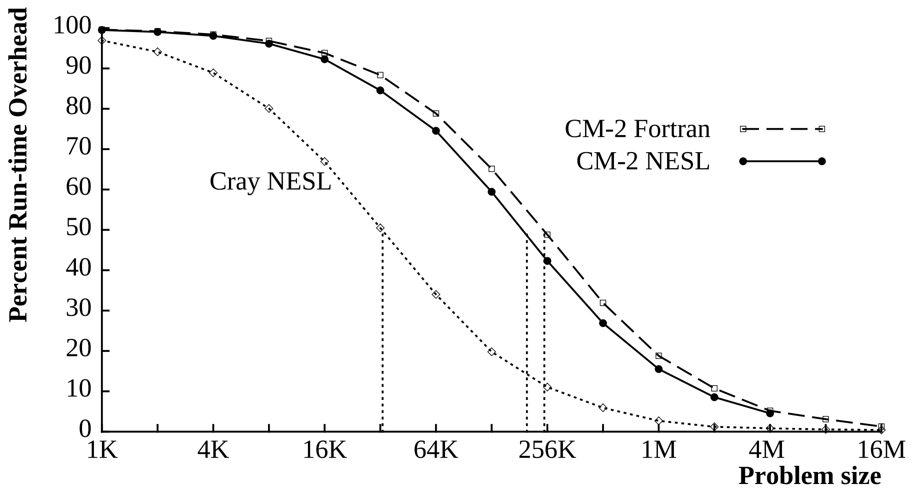

Figure 14: Interpreter overhead for the line-fitting benchmark. The vertical lines indicate the points at which overhead accounts for 50% of the running time. The percentage overhead for the CM-2 NESL implementation is comparable to that for the CM Fortran implementation. The Cray Fortran overhead is insignificant for the data sizes in the graph and is not shown.

6 Comparison to Other Systems

Numerous flat data-parallel languages have been proposed for portable parallel programming, such as C* [49], MPP-Pascal [7], *Lisp [39], UC [5], and Fortran 90 [2]. Section 2 explained some of the expressibility and efficiency limitations imposed by flat languages; these problems are also discussed elsewhere [10, 11, 29, 32, 54].

Two existing parallel languages permit the user to describe nested data-parallel operations: CM-Lisp [63] and Paralation Lisp [51]. However, the implementations of these languages only exploit the bottom level of parallelism; for the sparse-matrix example, this results in a parallel sum for each row and a serial loop over the rows. Both of these languages are data-parallel extensions to Common Lisp [55]. The large number of features in Common Lisp, and the difficulty of extending their semantics to parallel execution, preclude the implementation of full nested data parallelism. This is strong motivation for a simple core language.

The parallel languages ID [43], SISAL [40], and Crystal [22], although not explicitly data-parallel, do support fine-grained parallelism. They also support nested data structures, although there has been little research on implementing nested parallelism for these languages. There are also several serial languages that supply data-parallel primitives and nested structures; these include SETL [52], APL2 [37], J [36], and FP [4]. Sipelstein and Blelloch [54] discuss these languages from the perspective of supporting data parallelism.

Another approach to architecture-independent parallel programming is based on control-parallel languages that provide asynchronous communicating serial processes. Examples include CSP [35], Linda [18], Actors [1], and PVM [56]. These languages are well-suited for problems (including irregular problems) that can be specified in terms of coarse-grained sub-tasks. Unfortunately, high implementation overhead makes efficiency too dependent on finding a decomposition into reasonably sized blocks [18]. As a result, these systems are not well-suited for exploiting fine-grained parallelism. The large grain size renders programs less likely to be efficient on most parallel supercomputers because they will not vectorize well and do not expose enough parallelism to take advantage of large numbers of processors. Extending these models to capture fine-grained parallelism is an area of active research [23].

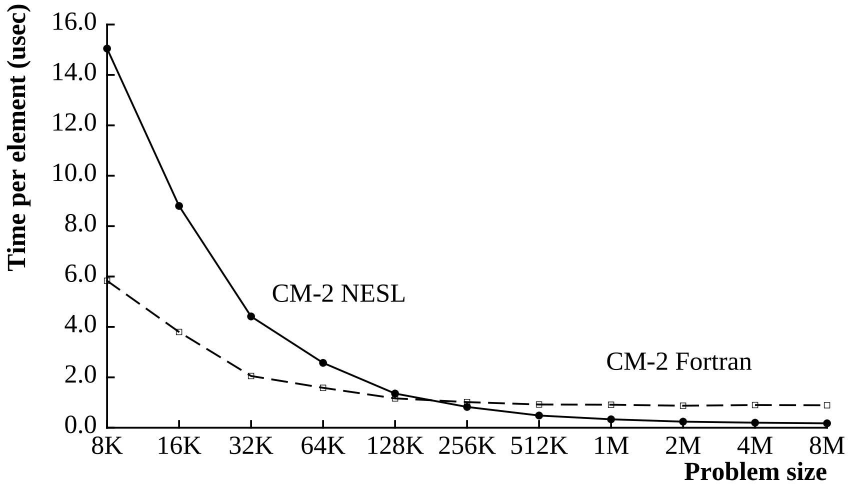

Figure 15: CM-2 median: NESL vs. CM Fortran.

7 Conclusions

The purpose of nested data-parallel languages is to provide the advantages of data parallelism while extending their applicability to algorithms that use "irregular" data structures. The main advantages of data parallelism that should be preserved are the efficient implementation of fine-grained parallelism and the simple synchronous programming model.

We have described the implementation of a nested data-parallel language called NESL. NESL was designed to allow the concise description of parallel algorithms on both structured and unstructured data. It has been used in a course on parallel algorithms and has allowed students to quickly implement a wide variety of programs, including systems for speech recognition, ray-tracing, volume rendering, parsing, maximum-flow, singular value decomposition, mesh partitioning, pattern matching, and big-number arithmetic [16]. (A full implementation of NESL is available from `blelloch@cs.cmu.edu`.)

The benchmark results in this paper show that it is possible to achieve good efficiency with a nested data-parallel language, across a variety of parallel machines. NESL runs within a local interactive environment that allows the user to execute programs remotely on any of the supported architectures. This portability depends crucially on the organization of the system and the use of an intermediate language.

The efficiency of NESL on large applications still requires further study. Some other issues that we plan to examine: getting better efficiency on nested parallel code with many conditionals; the specification of data layout for irregular structures; tools for profiling nested parallel code; the interaction of higher-order functions with nested parallelism; and porting the system to other architectures.

References

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[2] ANSI. *ANSI Fortran Draft S8, Version 111*.

[3] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, Cambridge, MA, 1991. ch. 4.

[4] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[5] R. Bagrodia and S. Mathur. Efficient implementation of high-level parallel programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 142–152, April 1991.

[6] Kenneth E. Batcher. The flip network of STARAN. In *Proceedings International Conference on Parallel Processing*, pages 65–71, 1976.

[7] Kenneth E. Batcher. The massively parallel processor system overview. In J. L. Potter, editor, *The Massively Parallel Processor*, pages 142–149. MIT Press, 1985.

[8] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[9] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[10] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[11] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[12] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.

[13] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[14] Guy E. Blelloch, Siddhartha Chatterjee, Fritz Knabe, Jay Sipelstein, and Marco Zagha. VCODE reference manual (version 1.1). Technical Report CMU-CS-90-146, School of Computer Science, Carnegie Mellon University, July 1990.

[15] Guy E. Blelloch, Siddhartha Chatterjee, and Marco Zagha. Solving linear recurrences with loop raking. In *Proceedings Sixth International Parallel Processing Symposium*, pages 416–424, March 1992.

[16] Guy E. Blelloch and Jonathan C. Hardwick. Class notes: Programming parallel algorithms. Technical Report CMU-CS-93-115, School of Computer Science, Carnegie Mellon University, February 1993.

[17] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[18] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[19] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1991.

[20] Siddhartha Chatterjee. Compiling nested data-parallel programs for shared memory multiprocessors. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1993.

[21] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, November 1990.

[22] Marina Chen, Y. Choo, and Jingke Li. Crystal: Theory and pragmatics of generating efficient parallel code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 7. Addison-Wesley, Reading, MA, 1991.

[23] Andrew A. Chien and William J. Dally. Experience with concurrent aggregates (CA): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*, pages 1040–1049. SIAM, April 1990.

[24] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual, SR-0085B*, March 1988.

[25] Cray Research Inc., Mendota Heights, Minnesota. *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual*, July 1989.

[26] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[27] Rickard E. Faith, Doug L. Hoffman, and David G. Stahl. UnCvl: The University of North Carolina C Vector Library. Version 1.1, May 1993.

[28] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[29] Geoffrey C. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-134, Syracuse Center for Computational Science, Syracuse University, 1991.

[30] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3.0 User's Guide and Refernce Manual*, February 1993.

[31] K. Gopinath and J. L. Hennessy. Copy elimination in functional languages. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 303–314, January 1989.

[32] Phil Hatcher, Walter F. Tichy, and Michael Philippsen. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[33] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[34] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[35] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[36] Roger K. W. Hui, Kenneth E. Iverson, E. E. McDonnell, and Arthur T. Whitney. APL\? In *APL 90 Conference Proceedings*, pages 192–200, Copenhagen, Denmark, August 1990.

[37] IBM. *APL2 Programming: Language Reference*, first edition, August 1984. Order Number SH20-9227-0.

[38] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report UW Technical Report #1126, Computer Science Department, University of Wisconsin-Madison, November 1992.

[39] Clifford Lasser. *The Essential *Lisp Manual*. Thinking Machines Corporation, Cambridge, MA, July 1986.

[40] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, March 1985.

[41] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[42] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[43] Rishiyur S. Nikhil. ID Version 90.0 Reference Manual. Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.

[44] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and Ch. Jacobi. The PASCAL P compiler: Implementation notes (revised edition). Technical Report 10, ETH, Switzerland, 10 1976.

[45] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, 1986.

[46] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[47] Michael J. Quinn and Philip J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, September 1990.

[48] Kay A. Robbins and Steven Robbins. *The Cray X-MP/Model 24: A Case Study in Pipelined Architecture and Vector Processing*. Springer-Verlag, 1989.

[49] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings Second International Conference on Supercomputing, Vol. 2*, pages 2–16, May 1987.

[50] Gary Sabot, Lisa Tennies, Alex Vasilevsky, and Richard Shapiro. Compiler parallelization of an elliptic grid generator for 1990 Gordon Bell Prize. In *Proceedings Supercomputing '91*, pages 338–346, November 1991.

[51] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, Massachusetts, 1988.

[52] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[53] Jay Sipelstein. *Data Representation Optimizations for Collection-Oriented Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University. To appear.

[54] Jay Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[55] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common LISP: The Language*. Digital Press, Burlington, MA, 1984.

[56] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[57] Thinking Machines Corporation, Cambridge, MA. *Paris Reference Manual*, February 1991.

[58] Thinking Machines Corporation, Cambridge, MA. *CDPEAC: Using GCC to program in DPEAC*, December 1992.

[59] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, December 1992.

[60] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, May 1993. Version 3.0.

[61] Thinking Machines Corporation, Cambridge, MA. *Programming the NI*, February 1993. Version 7.1.

[62] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[63] Skef Wholey and Guy L. Steele Jr. Connection Machine Lisp: A dialect of Common Lisp for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, May 1987.

[64] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

A C, Fortran, and CM Fortran Benchmark Code

C code for line fitting

```
void fit(double x[], double y[], double *a, double *b,
         double *siga, double *sigb, int n)
{
  int i;
  double chi2;
  double xa = 0.0;
  double ya = 0.0;
  double Stt = 0.0;

  *b = 0.0;
  for (i = 0; i < n; i++) {
    xa += x[i];
    ya += y[i];
  }
  xa = xa/n;
  ya = ya/n;
  for (i = 0; i < n; i++) {
    double tmp = x[i] - xa;
    Stt += tmp * tmp;
    *b += tmp * y[i];
  }
  *b = *b/Stt;
  *a = (ya - xa * *b);
  chi2 = 0;
  for (i = 0; i < n; i++) {
    double v = y[i] - *a - *b * x[i];
    chi2 += v * v;
  }
  *siga = sqrt((1.0/n + xa*xa/Stt)*chi2/(n-2.0));
  *sigb = sqrt((1.0/Stt)*chi2/(n-2.0));
}
```

Fortran code for line fitting

```
      subroutine fit(x, y, a, b, siga, sigb, n)
      implicit real (a-h, o-z)
      real x(n), y(n)

      xa = 0.0
      ya = 0.0
      do 10 j = 1, n
         xa = xa + x(j)
         ya = ya + y(j)
 10   continue
      xa = xa / n
      ya = ya / n
      Stt = 0.0
      b = 0.0
      do 30 j = 1, n
         tmp = x(j) - xa
         Stt = Stt + tmp * tmp
         b = b + tmp * y(j)
 30   continue
      b = b / Stt
      a = ya - xa*b
      chi2 = 0.0
      do 50 j = 1, n
         chi2 = chi2 + (y(j) - a - b*x(j))**2
 50   continue
      siga  = sqrt((1.0/n + xa*xa/Stt)*chi2/(n-2.0))
      sigb  = sqrt((1.0/Stt)*chi2/(n-2.0))
      return
      end
```

CM Fortran code for line fitting

```
      subroutine fit(x, y, a, b, siga, sigb, n)
      implicit double precision (a-h, o-z)
      double precision x(n), y(n)

      xa    = sum(x)/n
      ya    = sum(y)/n
      Stt   = sum((x - xa)**2)
      b     = sum((x - xa)*y)/Stt
      a     = ya - xa*b
      chi2  = sum((y - a - b*x)**2)
      siga  = sqrt((1.0/n + xa*xa/Stt)*chi2/(n-2.0))
      sigb  = sqrt((1.0/Stt)*chi2/(n-2.0))
      return
      end
```

C code for selection

```
int select(int s[], int k, int l, int *tos)
{
  int i;
  int pivot = s[l/2];
  int lesser = 0;
  for (i = 0; i < l; i++) if (s[i] < pivot) lesser++;
  if (k < lesser) {
    int j = 0;
    for (i = 0; i < l; i++) if (s[i] < pivot) tos[j++] = s[i];
    return select(tos, k, lesser, tos + lesser);
  }
  else {
    int greater = 0;
    for (i = 0; i < l; i++) if (s[i] > pivot) greater++;
    if (k >= l - greater) {
      int j = 0;
      for (i = 0; i < l; i++) if (s[i] > pivot) toss[j++] = s[i];
      return select(tos, k - (l - greater), greater, tos + greater);
    }
    else return pivot;
  }
}
```

Fortran code for selection

```
      integer function select(t, s, d, kk, ll)
      implicit integer (a-z)
      integer s(ll), t(ll), d(ll), pivot, greater, lesser
      k = kk
      l = ll
      do i = 1, l
         s(i) = d(i)
      end do

 5    continue
      pivot = s(l/2 + 1)
      lesser = 0
      do i = 1, l
         if (s(i) .lt.  pivot) lesser = lesser + 1
      end do
      if (k .lt.  lesser) then
         j = 0
         do i = 1, l
            if (s(i) .lt.  pivot) then
               j = j + 1
               t(j) = s(i)
            endif
         end do
         do i = 1, lesser
            s(i) = t(i)
         end do
         l = lesser
      else
         greater = 0
         do i = 1, l
            if (s(i) .gt.  pivot) greater = greater + 1
         end do
         if (k .ge.  l - greater) then
            j = 0
            do i = 1, l
               if (s(i) .gt.  pivot) then
                  j = j + 1
                  t(j) = s(i)
               endif
            end do
            do i = 1, greater
               s(i) = t(i)
            end do
            k = k - (l - greater)
            l = greater
         else
            l = 0
         endif
      endif
      if (l .gt.  0) goto 5
      select = pivot
      return
      end
```

CM Fortran code for selection

```
      integer function select(src, k, n)
      implicit integer (a-z)
      integer src(n), s(n)
      logical greater(n), lesser(n), mask(n)

      s = src
      on_count = n
      target = k
      do while (on_count .gt.  0)
         pivot = s(on_count/2 + 1)
         forall (i=1:n) mask(i) = i .le.  on_count
         lesser = s .lt.  pivot .and.  mask
         n_lesser = count(lesser)
         if (target .lt.  n_lesser) then
            s = pack(s, lesser)
            on_count = n_lesser
         else
            greater = s .gt.  pivot .and.  mask
            n_greater = count(greater)
            if (target .ge.  on_count - n_greater) then
               s = pack(s, greater)
               target = target - (on_count - n_greater)
               on_count = n_greater
            else
               on_count = 0
            endif
         endif
      end do
      select = pivot
      return
      end
```

C code for sparse-matrix vector product

```
void MxV(double Result[], double Mval[], int Midx[], int Mlen[],
 double Vect[], int nrows)
{
  int ncols;
  double sum;

  while (nrows--) {
    sum = 0.0;
    ncols = *Mlen++;
    while (ncols--) sum += *Mval++ * *(Vect + *Midx++);
    *Result++ = sum;
  }
}
```

Fortran code for sparse-matrix vector product

```
 subroutine MxV(Result, Mval, Midx, Mlen, Vect, nrows)
 integer nrows
 integer Midx(*), Mlen(*)
 real Result(*), Mval(*), Vect(*), sum

 index = 1
 do irow = 1, nrows
    sum = 0.0
    do icol = 1, Mlen(irow)
       sum = sum + Mval(index) * Vect(Midx(index))
       index = index + 1
    end do
    Result(irow) = sum
 end do
 return
 end
```

CM Fortran code for sparse-matrix vector product

```
 subroutine Setup(Mlen, Offset, Segment, nrows, n)
 include '/usr/include/cm/CMF_defs.h'
 integer nrows, n, Mlen(nrows), Offset(nrows)
 logical Segment(n)

 call CMF_SCAN_ADD(Offset, Mlen, CMF_NULL, 1,
$     CMF_UPWARD, CMF_EXCLUSIVE, CMF_NONE, .TRUE.)
 Offset = Offset + 1

 Segment = .FALSE.
 Segment(Offset) = .TRUE.
 return
 end


 subroutine MxV(Result,Mval,Midx,Offset,Segment,Vect,nrows,n)
 include '/usr/include/cm/CMF_defs.h'
 integer nrows, n, Midx(n), Offset(nrows)
 logical Segment(n)
 real Mval(n), ScanMe(n), Result(nrows), Vect(nrows)

 ScanMe = Mval * Vect(Midx)
 call CMF_SCAN_ADD(ScanMe, ScanMe, Segment, 1,
$     CMF_DOWNWARD, CMF_INCLUSIVE, CMF_SEGMENT_BIT, .TRUE.)
 Result = ScanMe(Offset)
 return
 end
```