

Implementation of a Portable Nested Data-Parallel Language*

Guy E. Blelloch¹ Siddhartha Chatterjee²

Jonathan C. Hardwick Jay Sipelstein Marco Zagha

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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, 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 Machine CM-2, the Cray Y-MP C90, and serial machines. 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 codes for regular dense data, and is often superior for irregular data.

* This research was sponsored 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. Cray Y-MP C90, Connection Machine CM-2 and Connection Machine CM-5 time was provided by the Pittsburgh Supercomputing Center (Grant ASC890018P).

¹ Partially supported by a Finmeccanica chair and an NSF Young Investigator award.

² RIACS, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94035

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 the U.S. government.

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 [23] and various extensions to C [35, 34, 4]. 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 ideally 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 [21]. 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 [45, 7, 22] combine aspects of both strict data-parallel languages and control-parallel languages. Nested data-parallel languages allow recursive data structures, and also the application of parallel functions to multiple sets of data in parallel. 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 all the rows of our sparse matrix. Because the nested calls are to the same function, a technique called *flattening nested parallelism* [13] allows a compiler to convert them into a form that runs 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

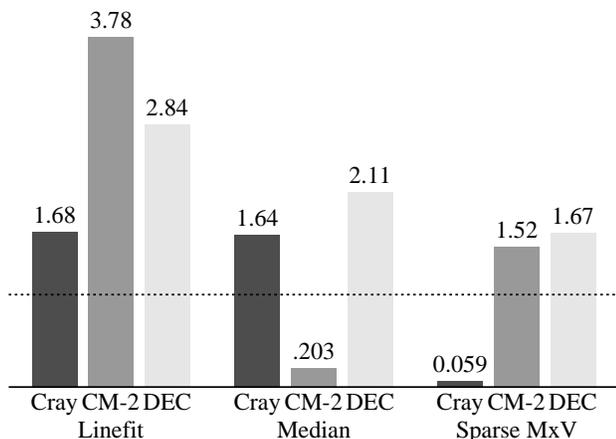


Figure 1: Performance summary for the three benchmarks. The numbers given are the ratio of the times taken by the NESL and native code versions of the benchmark (smaller numbers are therefore better). The problem size was the largest that fit in physical memory. Full performance results are given in Section 5.

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 Machine CM-2, the Cray C90, and serial workstations. (We are currently working on a version for the Connection Machine CM-5.) In this paper we describe the language and 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 we use 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 optimizations. 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 generates near-optimal code from FORTRAN 77 (vectorized) and CM Fortran. The other two benchmarks demonstrate the efficiency of dynamic memory allocation and nested-parallel code.

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 wider variety of algorithms, including computing the minimum-spanning-tree of sparse graphs, searching for patterns in strings, and finding the convex-hull of a set of points, is given elsewhere [12].) Section 3 outlines the different components of the current NESL implementation. Section 4 describes our benchmarks and Section 5 discusses the running times of the benchmarks.

2 NESL and Nested Parallelism

NESL is a high-level language designed to allow simple and concise descriptions of nested data-parallel programs [8]. It is strongly typed and applicative (side-effect-free). Sequences are the primitive parallel data type and the language provides a large set of built-in parallel sequence operations. Parallelism is achieved through the ability to apply functions in parallel to each element of a sequence. The application of a function over a sequence is specified using a set-like notation similar to *set-formers* in SETL [37] and *list-comprehensions* in Miranda [44] and Haskell [26]. For example, the NESL expression

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

can be read as “in parallel, for each \mathbf{a} in the sequence $[3, -4, -9, 5]$ such that \mathbf{a} is less than 4, negate \mathbf{a} ”. The expression returns $[-3, 4, 9]$. The parallelism is applied both to the expression to the left of the colon ($:$) and to the subselection to the right of the pipe ($|$). This parallel subselection can be implemented with packing techniques [5]. 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 [8]. These are similar to operations found in other data-parallel languages.

NESL supports nested parallelism by permitting sequences as elements of a sequence and by permitting the parallel sequence functions to be used in the apply-to-all 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
<code>#a</code>	<i>Length of sequence a.</i>
<code>a[i]</code>	<i>ith element of sequence a.</i>
<code>sum(a)</code>	<i>Sum of sequence a.</i>
<code>plus_scan(a)</code>	<i>Parallel prefix with addition.</i>
<code>permute(a,i)</code>	<i>Permute elements of sequence a to positions given in sequence i.</i>
<code>get(a,i)</code>	<i>Get values from sequence a based on indices in sequence i.</i>
<code>a ++ b</code>	<i>Append sequences a and b.</i>

Figure 2: Seven of NESL's sequence operations.

```

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};

Add 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 simple 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 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 similarly to sparse matrices, using each subsequence to keep an adjacency list.

Nested parallelism is also very useful in divide-and-conquer algorithms. As an example, consider a parallel quicksort algorithm (Figure 4). The three assignments

```

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.

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 function `++` appends the three sequences. In this algorithm, there is parallelism both within the pack required to select each of the three intermediate sequences and in the 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, due to the difficulty of implementing nested data-parallelism when nested function calls can interact with each other through side-effects. This problem can probably be overcome with aggressive compiler analysis to determine places where there is the possibility of interaction.

3 System Overview

Our implementation of NESL consists of an intermediate language called `VCODE` [9], a compiler [15, 16] and interpreter for `VCODE`, and a portable library of parallel routines called `CVL` [10]. Figure 5 illustrates how the implementation is organized. We split our system along these lines so that we could concentrate on different aspects of the system in isolation. This section gives an overview of the various components of the system: a more detailed description can be found in the full version of this paper [11].

VCODE: `VCODE` is a stack-based intermediate language where the objects on the stack are vectors of atomic values (integers, floats, or booleans). `VCODE` instructions act on these vectors as a whole, performing such operations as elementwise adding two vectors, summing a vector, or permuting the order of elements within a vector. To provide support for nested parallelism, `VCODE` supplies the notion of *segment descriptors* [7]. These objects, also kept on the stack, 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 never overlap). The `VCODE` representation of the segment descriptor is machine-dependent: our serial

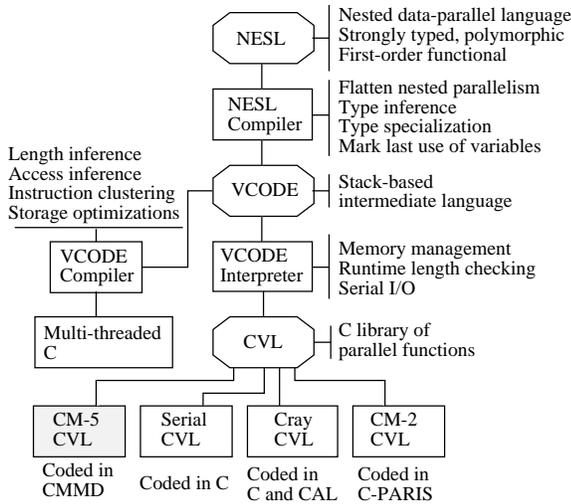


Figure 5: The parts of the SCAL project and how they fit together. CAL stands for Cray Assembly Language, and C-PARIS is the C interface to the Connection Machine Parallel Instruction Set. CM-5 CVL is under development.

implementation uses a sequence of the lengths of each segment, while our implementations on the Connection Machine CM-2 and Cray also use flags to mark boundaries between segments. Most of the non-elementwise VCODE instructions require both a vector and a segment descriptor as arguments. Each instruction then operates independently over each segment of the vector. For example, the `+_scan` instruction will perform 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. The notion of segments also appears in some of the library routines of the Connection Machines CM-2 and CM-5 [43, 42] and has been adopted in the PREFIX and SUFFIX operations of High Performance Fortran.

NESL compiler: The NESL compiler translates NESL code into VCODE. The most important compilation step is the use of a technique called *flattening nested parallelism* [13, 7]. This process converts nested sequences into sets of flat vectors and translates the nested operations into segmented VCODE operations on the flattened representation. The flattening of nested sequences is achieved by converting all sequences into a pair: a **value** vector and a **segdes** (segment descriptor). 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). 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 as multiple vectors (one for each slot in the structure) that each use a single segment descriptor.

Using this representation, nested versions of NESL operations with VCODE counterparts can be directly converted into the corresponding segmented VCODE instruction. Nested versions of user-defined functions are implemented by using program transformations called *stepping-up* and *stepping-down*. 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 called in parallel, each subcall gets placed in a separate segment, and the computations on these segments 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 subcalls that do not take a branch, and then reinserts this data when that branch is completed. This guarantees that work is only done on the subcalls that take the branch, and is similar to a technique used by vectorizing compilers to vectorize loops with conditionals. 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 minimized by only packing when there is a significant load imbalance among processors [38].

CVL: To enable us to quickly implement VCODE on new machines, we designed CVL (C Vector Library), a library of low-level segmented vector routines callable from C. These are used by a VCODE interpreter, described below. Our implementations of CVL on the Connection Machine CM-2, Cray and serial worksta-

tions are highly optimized. We originally tried to implement these operations in a high-level language, but to attain high efficiency, we were forced to use Cray Assembly Language (CAL) on the Cray and the Parallel Instruction Set (PARIS) on the Connection Machine CM-2. This was particularly important on the Cray, since we could not get the C or FORTRAN compilers to vectorize the segmented code (some of these issues are discussed in [17]). We might have achieved better performance on the Connection Machine CM-2 by going one level closer to the machine and using CMIS, but time did not permit this. Our implementation of CVL on workstations uses ANSI C and can therefore be ported easily to most serial machines.

VCODE interpreter: The two main requirements for the VCODE interpreter are portability and efficient management of the vector memory. The actual vector instructions supplied by VCODE are mapped into CVL calls in a fairly straightforward manner via table-lookup. The VCODE interpreter imposes a constant per executed instruction overhead on VCODE programs. Hence, efficient execution of these primitives within CVL guarantees efficient evaluation of VCODE programs for large vectors. The most interesting part of the interpreter is how it manages memory. The interpreter uses free lists of varying sized blocks of memory and reference counting on vectors in order to create and destroy vectors of differing and dynamically determined sizes. The algorithm used is fully described in [11].

VCODE compiler: Chatterjee’s doctoral dissertation [15] discusses the design, implementation, and evaluation of an optimizing VCODE compiler for shared-memory MIMD machines. 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 that limit both the asymptotic performance and the performance for small problem sizes. 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 compile-time knowledge of data sizes or the number of processors.

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

	Properties		
	Commu- nication	Dynamic Structures	Nested Parallel
Line Fit	Low	No	No
Median	High	Yes	No
Sparse MxV	High	No	Yes

Figure 6: The properties of the benchmarks.

```
function linefit(x, y) =
let
  n      = float(#x);
  xa     = sum(x)/n;
  ya     = sum(y)/n;
  Stt    = sum({(x - xa)^2: tmp});
  b      = sum({(x - xa)*y: tmp; 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 7: 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 $sigb$).

technology available for the machine.

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 6). 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 [31]. The performance of these benchmarks demonstrate many of the advantages and disadvantages of our system. The results of these benchmarks will be analyzed in Section 5.

Our first benchmark is a least-squares line-fitting routine using the algorithm described in Press *et al.* [33, section 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

```

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 8: 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 = get(Vect, Midx);
  p = {a * b: a in Mval; b in v}
in
  {sum(row) : row in nest(p, Mlen)};

```

Figure 9: 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`).

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 7.

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 [24]. This method is similar to quicksort, but calls itself recursively only on one of the two partitions, depending on which contains 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 re-distributed on each iteration. The NESL code for the algorithm is shown in Figure 8. This algorithm was selected to demonstrate the utility and efficiency of dynamic allocation.

Our third benchmark multiplies a sparse matrix by a

dense vector, and demonstrates the usefulness of nested parallelism. Sparse-matrix vector multiplication is an important supercomputer kernel 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 a challenge 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 long. 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 [20]. Figure 9 shows the NESL implementation.

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 [42] on the Connection Machine CM-2, and C on the DECstation 5000/20 (using the gcc compiler). In all cases we used full optimization, and in the case of the median-finding code on the Cray we had to include compiler directives (`ivdep`) to force vectorization. The full listing of the native code we used is given in [11]. NESL timings are for the code shown in the previous section run using the VCODE interpreter. 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 16K processors of a CM-2, each with 256 Kbits of memory. All DECstation benchmarks were run on a DECstation 5000/20 with 16 Mbytes of memory.

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 multiplication benchmark demonstrates the advantages and efficiency of nested data parallelism. Figure 10 gives running times on the Cray for a variety of degrees of

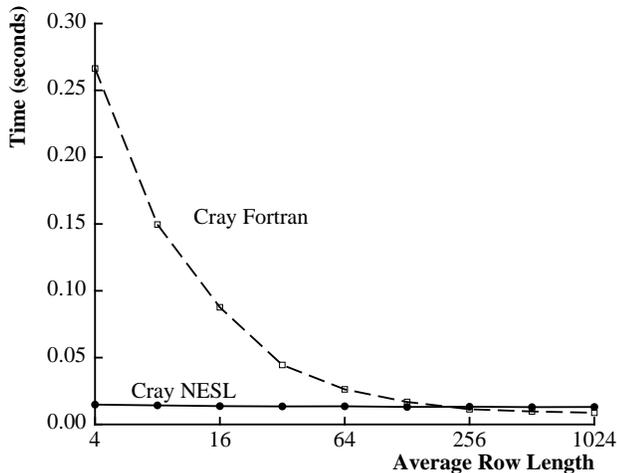


Figure 10: Running times of the sparse-matrix vector product for varying levels of sparseness. The number of nonzero entries in each sparse matrix is fixed at 10^6 .

sparseness. For very sparse matrices, the NESL version outperforms the native version by over a factor of ten. This is because the compilation of nested data parallelism described in Section 3 generates code with running time essentially independent of the size of the subsequences. The full efficiency (vectorization on the Cray and high data-to-processor ratio on the CM-2) of executing on the full input data is achieved by the nested code. The result is consistently high performance regardless of how sparse the matrix is. Note, however, that as the matrix density increases, the Cray FORTRAN performance improves. Eventually, FORTRAN achieves superior performance because of the 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 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 may be computed by examining the running times for small input. Figure 11 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 over-

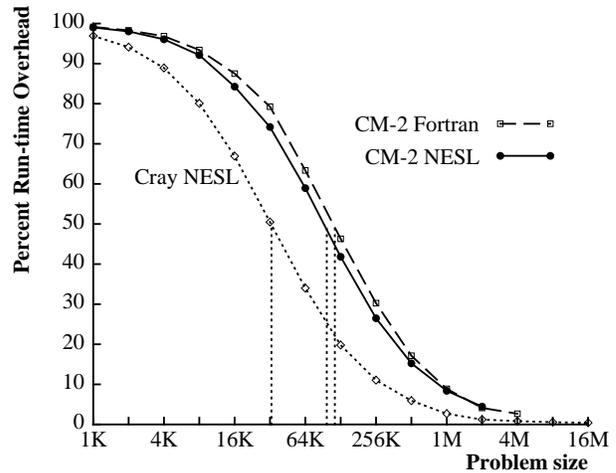


Figure 11: Interpreter overhead for the line-fitting benchmark. The vertical lines indicate the points at which overhead accounts for 50% of the running time. The percentage overhead for the CM-2 NESL implementation is comparable to that for the CM Fortran implementation. The Cray FORTRAN overhead is insignificant for the data sizes in the graph and is not shown.

head is not a problem on the CM-2; here, since there are 16K processors, the loss of efficiency when working with small vectors ($n < 16K$) 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 inefficiencies adversely affect the peak performance of NESL programs, and the effects can be seen in the performance of line-fitting for large data sizes (see Table 1).¹

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

¹On the CM-2 there is an additional important source of inefficiency: CM-2 CVL is built on top of the Paris instruction set [43]. Although working with Paris has many advantages, it forces use of the older “fieldwise” representation of data, instead of the more efficient “slice-wise” representation generated by the CM Fortran compiler.

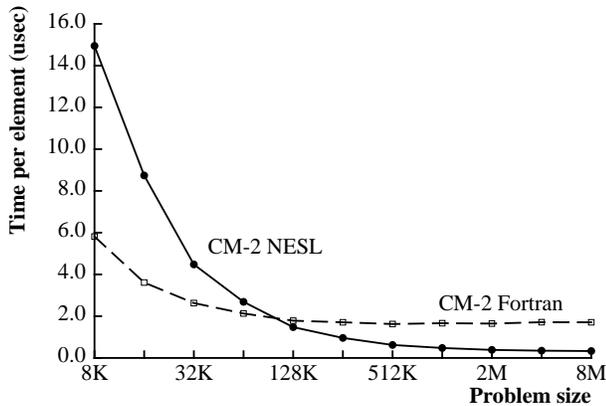


Figure 12: CM-2 median: NESL vs. CM Fortran.

n	DEC C	DEC NESL	C90 F77	C90 NESL	CM-2 CMF	CM-2 NESL
Line Fit						
2^{10}	0.0048	0.0156	0.0001	0.0012	0.0018	0.0061
2^{14}	0.1094	0.2969	0.0004	0.0018	0.0019	0.0061
2^{18}	1.7500	4.9844	0.0058	0.0122	0.0055	0.0208
2^{22}			0.0927	0.1551	0.0633	
Median						
2^{10}	0.0024	0.0156	0.0001	0.0059	0.0266	0.1009
2^{14}	0.0508	0.1094	0.0005	0.0092	0.0592	0.1432
2^{18}	1.0156	2.1406	0.0080	0.0233	0.4479	0.2515
2^{22}			0.1276	0.2099	7.1841	1.4565
Sparse Matrix-Vector Multiply						
2^{10}	0.0156	0.0156	0.0002	0.0003	0.0043	0.0137
2^{14}	0.0469	0.0625	0.0037	0.0006	0.0072	0.0161
2^{18}	0.8750	1.4688	0.0589	0.0038	0.0541	0.0823
2^{22}			0.9436	0.0557	0.8020	

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.

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 Fortran. In NESL, vectors are dynamically allocated with the data automatically balanced across the processors. Although the median algorithm requires $O(\log n)$ steps on average, the NESL implementation only requires a total of $O(n)$ work because on each step 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 for large inputs, as illustrated in Figure 12.

6 Comparison to Other Systems

Numerous flat data-parallel languages have been proposed for portable parallel programming, such as C* [35], MPP-Pascal [6], *Lisp [29], UC [4], and FORTRAN 90 [2]. Section 2 explained some of the expressibility and efficiency limitations imposed by this type of language. These problems are also discussed elsewhere [7, 21, 39, 22, 8].

There are two existing languages that permit the user to describe nested data-parallel operations: Connection Machine Lisp [45] and Paralation Lisp [36]. However, the implementations of these languages are only able to 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 these languages are data-parallel extensions to Common Lisp, and the difficulty of extending their semantics to parallel execution, preclude the implementation of full nested data parallelism. This is the main reason why we wanted a simple core language.

The parallel languages ID [32], SISAL [30], and Crystal [18], 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 [37], APL2 [28], J [27], and FP [3]. For a discussion of these languages from the perspective of supporting data parallelism see [39].

Another approach to architecture-independent parallel programming is control-parallel languages that provide asynchronous communicating serial processes. Examples include CSP [25], Linda [14], Actors [1], and PVM [41]. These languages are well suited for problems (including irregular problems) that can be specified in terms of coarse-grained sub-tasks. Unfortunately, high overhead in the implementation makes efficiency too dependent on finding a decomposition into reasonably sized blocks [14]. 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 won't vectorize well and don't expose enough parallelism to take advantage of large numbers of processors. Extending this model to capture fine-grained parallelism is an area of active research [19].

7 Conclusions

The purpose of nested data-parallel languages is to provide the advantages of data parallelism while ex-

tending 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 [12].² The benchmark results in this paper have shown that it is possible to get good efficiency with a nested data-parallel language, across a variety of different 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: (1) getting good efficiency on nested parallel code with many conditionals, (2) the specification of data layout for irregular structures, (3) tools for profiling nested parallel code, (4) the interaction of higher-order functions with nested parallelism, and (5) porting the system to other architectures.

References

- [1] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, Sept. 1990.
- [2] ANSI. *ANSI Fortran Draft S8, Version 111*.
- [3] 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, Aug. 1978.
- [4] 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 142–52, Apr. 1991.
- [5] K. E. Batcher. The flip network of STARAN. In *Proceedings International Conference on Parallel Processing*, pages 65–71, 1976.
- [6] K. E. Batcher. The massively parallel processor system overview. In J. L. Potter, editor, *The Massively Parallel Processor*, pages 142–149. MIT Press, 1985.
- [7] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [8] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, Jan. 1992.
- [9] G. E. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings Frontiers of Massively Parallel Computation*, pages 471–480, Oct. 1990.
- [10] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zaghera. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, Feb. 1993.
- [11] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zaghera. Implementation of a portable nested data-parallel language. Technical Report CMU-CS-93-112, School of Computer Science, Carnegie Mellon University, Feb. 1993.
- [12] G. E. Blelloch and J. C. Hardwick. Class notes: Programming parallel algorithms. Technical Report CMU-CS-93-115, School of Computer Science, Carnegie Mellon University, Feb. 1993.
- [13] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, Feb. 1990.
- [14] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989.
- [15] S. Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, Oct. 1991.
- [16] S. Chatterjee, G. E. Blelloch, and A. L. Fisher. Size and access inference for data-parallel programs. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 130–144, June 1991.
- [17] S. Chatterjee, G. E. Blelloch, and M. Zaghera. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, Nov. 1990.

²A full implementation of NESL is available from blelloch@cs.cmu.edu.

- [18] M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 7. Addison-Wesley, Reading, MA, 1991.
- [19] A. A. Chien and W. J. Dally. Experience with concurrent aggregates (CA): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*. SIAM, Apr. 1990.
- [20] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.
- [21] G. C. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-134, Syracuse Center for Computational Science, Syracuse University, 1991.
- [22] P. Hatcher, W. F. Tichy, and M. Philippsen. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.
- [23] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Jan. 1993.
- [24] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.
- [25] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.
- [26] P. Hudak and P. Wadler. Report on the functional programming language HASKELL. Technical report, Yale University, Apr. 1990.
- [27] R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney. APL\?. In *APL 90 Conference Proceedings*, pages 192–200, Jan. 1990.
- [28] IBM. *APL2 Programming: Language Reference*, first edition, Aug. 1984. Order Number SH20-9227-0.
- [29] C. Lasser. *The Essential *Lisp Manual*. Thinking Machines Corporation, Cambridge, MA, July 1986.
- [30] J. McGraw, S. Skedzielewski, S. Allan, R. Oldhoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, Mar. 1985.
- [31] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986.
- [32] R. S. Nikhil. ID Version 90.0 Reference Manual. Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.
- [33] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, 1986.
- [34] M. J. Quinn and P. J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, Sept. 1990.
- [35] 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.
- [36] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, Massachusetts, 1988.
- [37] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.
- [38] J. Sipelstein. *Data Representation Optimizations for Collection-Oriented Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, To appear.
- [39] J. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, Apr. 1991.
- [40] G. L. Steele Jr., S. E. Fahlman, R. P. Gabriel, D. A. Moon, and D. L. Weinreb. *Common LISP: The Language*. Digital Press, Burlington, MA, 1984.
- [41] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.
- [42] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, July 1991.
- [43] Thinking Machines Corporation, Cambridge, MA. *Paris Reference Manual*, Feb. 1991.
- [44] D. Turner. An overview of MIRANDA. *SIGPLAN Notices*, Dec. 1986.
- [45] S. Wholey and G. L. Steele Jr. Connection Machine Lisp: A dialect of Common Lisp for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, May 1987.