JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 21, 4–14 (1994)

Implementation of a Portable Nested Data-Parallel Language*,†

GUY E. BLELLOCH,‡,§,¶ JONATHAN C. HARDWICK,‡ JAY SIPELSTEIN,‡ MARCO ZAGHA,‡ AND SIDDHARTHA CHATTERJEE‖

‡*School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213; and* ‖*RIACS, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, California 94035*

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 Machines CM-2 and CM-5, the Cray Y-MP 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 codes for regular dense data, and is often superior for irregular data. © 1994 Academic Press, Inc.

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 [28] and various extensions to C (such as C* [40, 39], UC [5], and C** [33]). 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 [24]. 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 [8] 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* [14] 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 better 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

* 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 7597. Cray Y-MP C90 and Connection Machine CM-2 time was provided by the Pittsburgh Supercomputing Center (Grant ASC890018P). Connection Machine CM-5 time was provided by the National Center for Supercomputing Applications (Grant TRA930102N). 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.

† A preliminary version of this paper appeared in the Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, May 1993.

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

¶ To whom correspondence should be addressed at School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213. E-mail address: blelloch@cs.cmu.edu.

0743-7315/94 $6.00

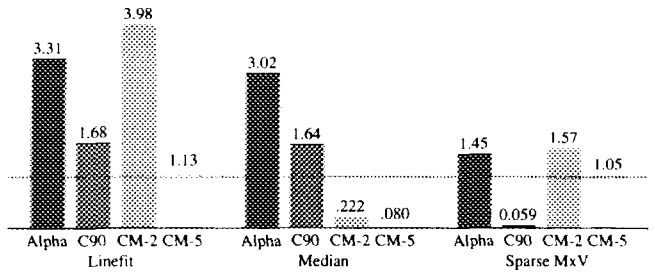

FIG. 1. Performance summary for the three benchmarks. The numbers given are the relative performance of NESL and native code versions of the benchmark (smaller numbers are therefore better) on large data sets. Full performance results are given in Section 5.

the Connection Machines CM-2 and CM-5, the Cray C90, and serial workstations. (We are currently working on a version for clusters of workstations.) 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, 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 [13].) 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 [9]. 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 [42]. 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 [9]. 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)]].
```

This representation can be used for matrices with arbitrary patterns of nonzero 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

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.
`get(a,i)`	*Get values from sequence* a *based on indices in sequence* i.
`a ++ b`	*Append sequences* a *and* b.

FIG. 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);
```

FIG. 3. Some simple operations on sparse matrices. Note that the last operation permutes whole rows.

proportional to the total number of nonzero 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 (Fig. 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 of the difficulty of implementing (and defining consistent semantics for) nested data-parallelism when nested function calls can interact with each other through side effects.

```
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];
```

FIG. 4. A nested data-parallel quicksort in NESL.

Although we feel that it would be possible to add nested data-parallelism to an imperative language, we 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 get good efficiency. 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 large number of 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 an intermediate language called VCODE [10], an interpreter for VCODE, and a portable library of parallel routines called CVL [12]. We also have an experimental VCODE compiler, described elsewhere [16, 17]. Figure 5 illustrates the roles of the different components of the implementation. This section gives an overview of 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 re-

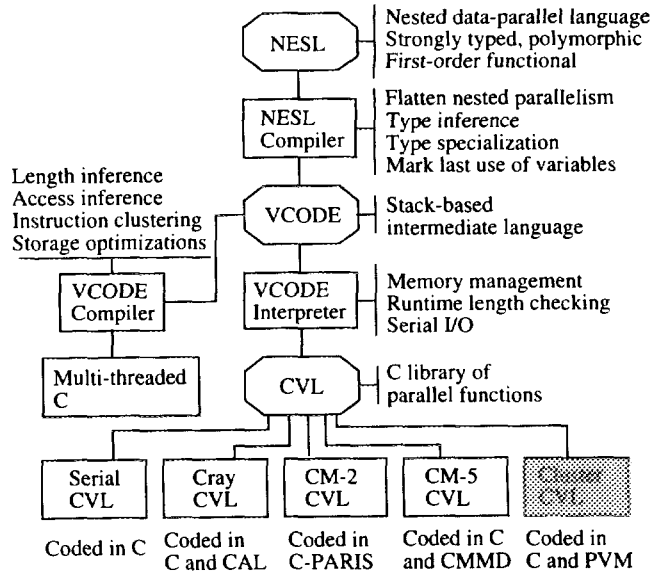

FIG. 5. The parts of the SCAL project and how they fit together. CAL stands for Cray Assembly Language, C-PARIS is the C interface to the CM-2 Parallel Instruction Set, and CMMD is the CM-5 message-passing library. Cluster CVL is under development.

written 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. 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 the elements of a vector, or permuting the order of elements within a vector. To provide support for nested parallelism, VCODE supplies the notion of *segment descriptors* [8]. 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 non-overlapping). The 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. Most of the non-elementwise VCODE instructions require both vector and segment descriptor arguments. 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 3.2 below). The notion of segments also appears in some of the library routines of the Connection Machines CM-2 [47] and CM-5 [49] and has been adopted in the PREFIX and SUFFIX operations of High Performance Fortran [28].

3.2. NESL *Compiler*

The NESL compiler translates NESL code into VCODE. The most import compilation step is the use of a technique called *flattening nested parallelism* [8, 14]. 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 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 datum is critical when nested versions of user-defined functions are implemented). 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 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 instructions. 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 used in an apply-to-each form, the data for each subcall are 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 these 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 [51, section 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 [43].

3.3. CVL

To enable rapid porting of VCODE to 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 in Section 3.4 below. Our implementations of CVL on the Connection Machine CM-2, Cray C90, and serial workstations are highly optimized. We originally tried to implement the CVL 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. Using CAL was particularly important on the Cray, since we could not get the C or FORTRAN compilers to vectorize the functions that operate on segmented data (some of the issues involved are discussed in [18]). 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. Our current implementation on the Connection Machine CM-5 is an initial release that uses C and CMMD 3.0 [49], and does not exploit the vector units. Cluster CVL is being written using C and PVM [25]. Faith *et al.* [22] at the University of North Carolina have ported CVL to the Maspar MP-2 using the MPL programming language.

3.4. 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 stemming 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; after completion, the arguments are popped off the stack. All this results in a constant interpretive overhead time 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 (for example, [3]) assume 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.

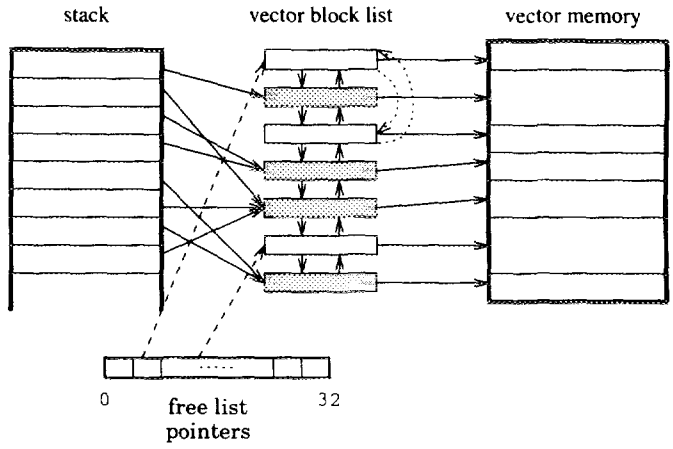

FIG. 6. Internal structure of the VCODE interpreter. The dark entries in the vector block list refer to active regions of vector memory. The light ones correspond to free regions. Entries to free regions of similar size are linked together in the same free list structure. There are 32 free lists, each one referring to regions at most twice the size of the previous.

Figure 6 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 Fig. 6, 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*

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

FIG. 7. The properties of the benchmarks.

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 [23, 26]. 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.

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 Fig. 7). 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 [36]. 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.* [38, section 14.2]. The version we use assumes that 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 Fig. 8.

Our second benchmark is a median-finding algorithm. To implement median-finding we use a more general algorithm that finds the kth smallest element in a set. The algorithm is based on the quickselect method [29]. 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 Fig. 9. 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 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 signifi-

```
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);
```

FIG. 8. 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`).

```
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);
```

FIG. 9. NESL code for median finding. The function `select_kth` returns the kth smallest element of `s`. This is used by `median` to find the middle element.

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

```
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]
```

FIG. 10. An example of sparse-matrix vector product.

```
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)};
```

FIG. 11. 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`).

cantly 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 nonzero elements in each row, and the values of each nonzero matrix element along with its column index [21]. Figure 10 shows an example of a sparse-matrix vector product using this format and Fig. 11 shows the NESL implementation.

5. RESULTS

Running times for all benchmarks with a variety of data sizes are given in Table I. Times are given both for interpreted NESL code and for native code. For native code we used FORTRAN 77 on the Cray C90, CM Fortran [48] on the Connection Machines CM-2 and CM-5, and C on the DEC Alpha workstation. 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 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 1 Gbyte of memory. All Connection Machine CM-5 benchmarks were run on 256 processors of a CM-5 with 8 Gbytes of memory.

The CM-5 CM Fortran benchmarks did not use the vector units, to allow a better comparison 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 times faster. It is expected that a future version of CM-5 CVL will exploit the vector units and will therefore achieve similar speedups.

We now discuss three main issues exhibited by the

TABLE I
Running Times in (Seconds) of the Benchmarks for NESL and Native Code

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	0.0063
2^{14}	0.0137	0.0468	0.0004	0.0018	0.0019	0.0061	0.0011	0.0063
2^{18}	0.2869	0.9506	0.0058	0.0122	0.0037	0.0133	0.0057	0.0095
2^{22}			0.0927	0.1551	0.0322	0.1283	0.1473	0.1658
				Median				
2^{10}	0.0004	0.0059	0.0001	0.0059	0.0293	0.1017	0.0086	0.0376
2^{14}	0.0068	0.0273	0.0005	0.0092	0.0623	0.1442	0.0215	0.0544
2^{18}	0.1347	0.4070	0.0080	0.0233	0.2667	0.2163	0.2146	0.0945
2^{22}			0.1276	0.2099	3.7810	0.8389	8.2092	0.6564
				Sparse-matrix vector product				
2^{10}	0.0002	0.0009	0.0002	0.0003	0.0043	0.0142	0.0012	0.0012
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	0.0259
2^{22}			0.9436	0.0557	0.4098	0.6436	0.2791	0.2929

Note. The sparse-matrix vector product uses a row length of 5 and randomly selected column indices. CM-5 NESL results are preliminary.

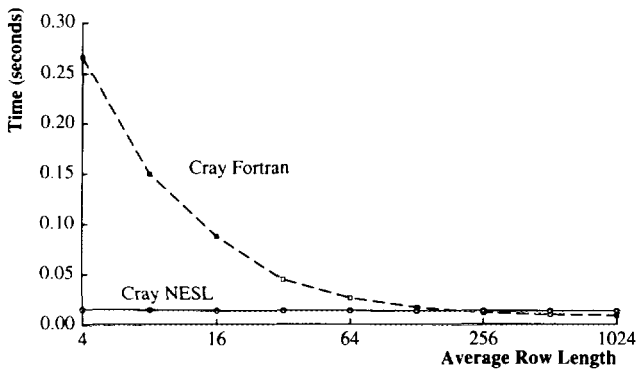

FIG. 12. Running times of the sparse-matrix vector product for varying levels of sparsity. The number of nonzero entries in each sparse matrix is fixed at 10^6.

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 12 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 10. We get this performance gain because the compilation of nested data parallelism described in Section 3.2 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 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 13 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; here, since there are 32K processors, the loss of efficiency when working with small vectors ($n < 32$ K) 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 [16, 17]. 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 I). On the CM-2 there is an additional important source of inefficiency: CM-2 CVL is built on top of the Paris instruction set [47]. 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 Fortran. In NESL,

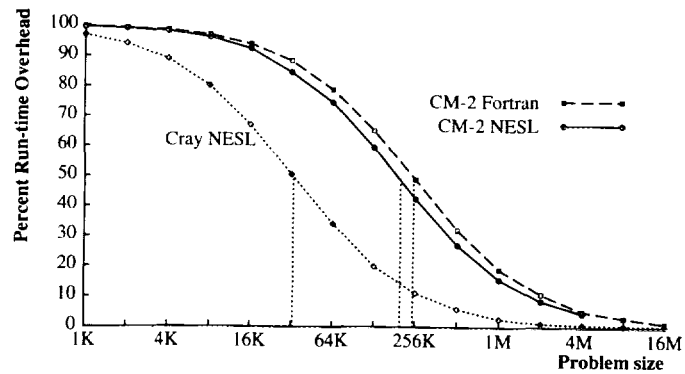

FIG. 13. Interpreter overhead for the line-fitting benchmark. The vertical lines indicate the points at which overhead accounts for 50% of the running time. The percentage overhead for the CM-2 NESL implementation is comparable to that for the CM Fortran implementation. The Cray FORTRAN overhead is insignificant for the data sizes in the graph and is not shown.

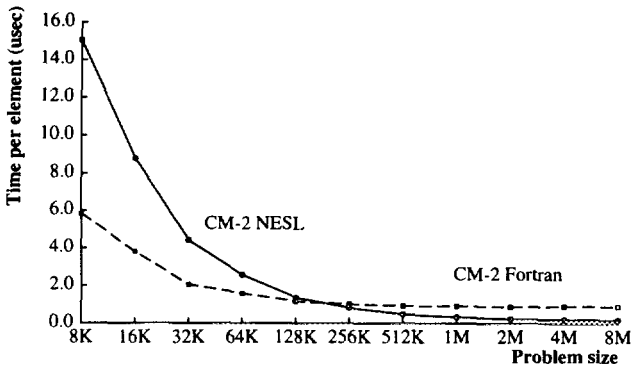

FIG. 14. CM-2 median: NESL vs CM Fortran.

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 Fig. 14.

6. COMPARISON TO OTHER SYSTEMS

Numerous flat data-parallel languages have been proposed for portable parallel programming, such as C* [40], MPP-Pascal [7], *Lisp [34], 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 [8, 9, 24, 27, 44].

Two existing parallel languages permit the user to describe nested data-parallel operations: Connection Machine Lisp [50] and Paralation Lisp [41]. 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 these languages are data-parallel extensions to Common Lisp [45]. 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 [37], SISAL [35], and Crystal [19], 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 [42], APL2 [32], J [31], and FP [4]. Sipelstein and Blelloch [44] discuss these languages from the perspective of supporting data parallelism.

Another approach to architecture-independent parallel programming is control-parallel languages that provide asynchronous communicating serial processes. Examples include CSP [30], Linda [15], Actors [1], and PVM [46]. These languages are well suited for problems (including irregular problems) that can be specified in terms of coarse-grained subtasks. Unfortunately, high implementation overhead makes efficiency too dependent on finding a decomposition into reasonably sized blocks [15]. 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 [20].

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 [13]. (A full implementation of NESL is available from `nesl-request@cs.cmu.edu`.)

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 are (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. Agha, G. Concurrent object-oriented programming. *Comm. ACM* **33**, 9 (Sept. 1990), 125–141.
2. ANSI. *ANSI Fortran Draft S8, Version 111.*

3. Appel, A. W. Garbage collection. In Lee, P. (Ed.), *Topics in Advanced Language Implementation*, MIT Press, Cambridge, MA, 1991, Chap. 4.
4. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* **21,** 8 (Aug. 1978), 613–641.
5. Bagrodia, R., and Mathur, S. Efficient implementation of high-level parallel programs. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1991, pp. 142–152.
6. Batcher, K. E. The flip network of STARAN. *Proc. International Conference on Parallel Processing*. 1976, pp. 65–71.
7. Batcher, K. E. The massively parallel processor system overview. In Potter, J. L. (Ed.), *The Massively Parallel Processor*. Cambridge, MA, MIT Press, 1985, pp. 142–149.
8. Blelloch, G. E. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.
9. Blelloch, G. E. NESL: A nested data-parallel language (version 2.6). Tech. Rep. CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Apr. 1993.
10. Blelloch, G. E., and Chatterjee, S. VCODE: A data-parallel intermediate language. In *Proc. Frontiers of Massively Parallel Computation*. 1990, pp. 471–480.
11. Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J., and Zagha, M. Implementation of a portable nested data-parallel language. Tech. Rep. CMU-CS-93-112, School of Computer Science, Carnegie Mellon University, 1993.
12. Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J., and Zagha, M. CVL: A C vector library. Tech. Rep. CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, 1993.
13. Blelloch, G. E., and Hardwick, J. C. Class notes: Programming parallel algorithms. Tech. Rep. CMU-CS-93-115, School of Computer Science, Carnegie Mellon University, 1993.
14. Blelloch, G. E., and Sabot, G. W. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.* **8,** 2 (Feb. 1990), 119–134.
15. Carriero, N., and Gelernter, D. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv.* **21,** 3 (Sept. 1989), 323–357.
16. Chatterjee, S. Compiling data-parallel programs for efficient execution on shared-memory multiprocessors. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, 1991.
17. Chatterjee, S. Compiling nested data-parallel programs for shared memory multiprocessors. *ACM Trans. Programming Languages Systems*. **15,** 3 (July 1993), 400–462.
18. Chatterjee, S., Blelloch, G. E., and Zagha, M. Scan primitives for vector computers. In *Proceedings Supercomputing '90*. 1990, pp. 666–675.
19. Chen, M., Choo, Y., and Li, J. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski (Ed.), *Parallel Functional Languages and Compilers*. Addison–Wesley, Reading, MA, 1991, Chap. 7.
20. Chien, A. A., and Dally, W. J. Experience with concurrent aggregates (CA): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*. SIAM, 1990, pp. 1040–1049.
21. Duff, I. S., Grimes, R. G., and Lewis, J. G. Sparse matrix test problems. *ACM Trans. Math. Software* **15** (1989), 1–14.
22. Faith, R. E., Hoffman, D. L., and Stahl, D. G. UnCvl: The University of North Carolina C Vector Library. Version 1.1, May 1993.
23. Feo, J. T., Cann, D. C., and Oldehoeft, R. R. A Report on the Sisal Language Project. *J. Parallel Distrib. Comput.* **10,** 4 (Dec. 1990), 349–366.
24. Fox, G. C. The architecture of problems and portable parallel software systems. Tech. Rep. SCCS-134, Syracuse Center for Computational Science, Syracuse University, 1991.
25. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM 3.0 User's Guide and Reference Manual*. 1993.
26. Gopinath, K., and Hennessy, J. L. Copy elimination in functional languages. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*. 1989, pp. 303–314.
27. Hatcher, P., Tichy, W. F., and Philippsen, M. A critique of the programming language C*. *Comm. ACM* **35,** 6 (June 1992), 21–24.
28. High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.
29. Hoare, C. A. R. Algorithm 63 (partition) and algorithm 65 (find). *Comm. ACM* **4,** 7 (1961), 321–322.
30. Hoare, C. A. R. Communicating sequential processes. *Comm. ACM* **21,** 8 (Aug. 1978), 666–677.
31. Hui, R. K. W., Iverson, K. E., McDonnell, E. E., and Whitney, A. T. APL\? *APL 90 Conference Proceedings*. 1990, pp. 192–200.
32. IBM. *APL2 Programming: Language Reference*, first ed., 1984. [Order Number SH20-9227-0].
33. Larus, J. R., Richards, B., and Viswanathan, G. C**: A large-grain, object-oriented, data-parallel programming language. Tech. Rep. UW Technical Report #1126, Computer Science Department, University of Wisconsin-Madison, Nov. 1992.
34. Lasser, C. *The Essential *Lisp Manual*. Thinking Machines Corporation, Cambridge, MA, 1986.
35. McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B., and Thomas, R. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, Mar. 1985.
36. McMahon, F. H. The Livermore Fortran kernels: A computer test of the numerical performance range. Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986.
37. Nikhil, R. S. ID Version 90.0 Reference Manual. Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.
38. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes*. Cambridge Univ. Press, Cambridge, 1986.
39. Quinn, M. J., and Hatcher, P. J. Data-parallel programming on multicomputers. *IEEE Software* **7,** 5 (Sept. 1990), 69–76.
40. Rose, J. R., and Steele, Jr., G. L. C*: An extended C language for data parallel programming. *Proceedings, Second International Conference on Supercomputing*. 1987, Vol. 2, pp. 2–16.
41. Sabot, G. W. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, Cambridge, Massachusetts, 1988.
42. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg, E. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.
43. Sipelstein, J. Data representation optimizations for collection-oriented languages. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, to appear.
44. Sipelstein, J., and Blelloch, G. E. Collection-oriented languages. *Proc. IEEE* **79,** 4 (Apr. 1991), 504–523.
45. Steele, G. L., Jr., Fahlman, S. E., Gabriel, R. P., Moon, D. A., and Weinreb, D. L. *Common LISP: The Language*. Digital Press, Burlington, MA, 1984.
46. Sunderam, V. S. PVM: A framework for parallel distributed computing. *Concurrency: Practice Experience* **2,** 4 (Dec. 1990), 315–339.

47. Thinking Machines Corporation. *Paris Reference Manual.* Cambridge, MA, 1991.
48. Thinking Machines Corporation. *CM Fortran Reference Manual.* Cambridge, MA, 1992.
49. Thinking Machines Corporation. *CMMD Reference Manual.* Cambridge, MA, 1993.
50. Wholey, S., and Steele G. L., Jr., Connection Machine Lisp: A dialect of Common Lisp for data parallel programming. *Proceedings, Second International Conference on Supercomputing.* 1987.
51. Wolfe, M. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

GUY BLELLOCH is an associate professor in the School of Computer Science at Carnegie Mellon University. His research is concerned with practical issues in the design and implementation of parallel algorithms for highly parallel computers. Issues of interest include parallel primitives, languages, compilers and implementation techniques. Blelloch received his B.A. (1983) in physics from Swarthmore College, and his M.S. (1986) and Ph.D. (1988) in computer science from the Massachusetts Institute of Technology.

SIDDHARTHA CHATTERJEE is a Postdoctoral Scientist at the Research Institute for Advanced Computer Science (RIACS) in Moffett Field, California. He has published papers in the areas of compilers for parallel languages, computer architecture, and parallel algorithms. His research interests include the design and implementation of parallel programming languages, high performance parallel architectures, and parallel algorithms and applications. Chatterjee received his B. Tech. (1985) in electronics and electrical communications engineering from the Indian Institute of Technology, Kharagpur, and his M.S. (1988) and Ph.D. (1991) in computer science from Carnegie Mellon University.

JONATHAN HARDWICK is a doctoral candidate in the School of Computer Science at Carnegie Mellon University. His research interests include the implementation of parallel languages on loosely coupled parallel machines. Hardwick received his B.A. (1989) in computer science from the University of Cambridge.

JAY SIPELSTEIN is a doctoral candidate in the School of Computer Science at Carnegie Mellon University. His research is focused on the development of compiler optimizations for collection-oriented languages. Sipelstein received his B.S. (1987) in mathematics from Yale University.

MARCO ZAGHA is a doctoral candidate in the School of Computer Science at Carnegie Mellon University. His research interests include parallel algorithms for unstructured and combinatorial problems (such as sparse matrix computation and sorting) and parallel language implementation. Zagha received his B.S. (1988) in computer science and engineering from the University of California at Los Angeles and his M.S. (1991) in computer science from Carnegie Mellon University.

Received March 10, 1993; accepted October 26, 1993