# USER-DEFINED REDUCTIONS FOR COMMUNICATION IN DATA-PARALLEL LANGUAGES

Guhan Viswanathan
James R. Larus

Technical Report #1293

January 1996

# User-defined Reductions for Communication in Data-Parallel Languages*

Guhan Viswanathan James R. Larus
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
Telephone: (608) 262-2542
{gviswana,larus}@cs.wisc.edu

January 12, 1996

## 1 Introduction

Parallel programming and parallel computers, have been a gleam in the eye of computer science for three or four decades. Rapid advances in semiconductor technology have led to high-performance, low-cost microprocessors that are appropriate components for a parallel machine. Unfortunately, this progress has left parallel software far behind. The difficulty of programming parallel computers is now, by far, the largest obstacle to their widespread use.

Improved parallel programming languages could reduce the difficulty of programming parallel computers by making programs less error prone and less machine specific. One promising approach is data-parallel languages, such as HPF [9], C* [18], or NESL [2], which provide high-level abstractions for the three key facets of parallel programming: concurrency, synchronization, and communication. In these languages, programmers express parallelism by invoking a parallel operation simultaneously on a collection of data. Synchronization is implicit in the division of a program into sequential and data-parallel phases.

The widest divergence among these languages is among their features for parallel communication. These languages pass values between sequential and parallel phases through arguments to parallel functions and, in most languages, by permitting these functions to reference data in a global address space. More interesting and varied are the mechanisms for passing values from the parallel to the sequential computations. Vector-based languages, such as Connection Machine Lisp [23] or NESL [2], provide vector permutation and mapping operations that rearrange a vector's data. Point-based languages, such as HPF [9] and C* [18],

*This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF PYI/NYI Awards CCR-9157366, MIPS-8957278, and CCR-9357779, NSF Grant MIP-9225097, DOE Grant DE-FG02-93ER25176, University of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

use assignments in a global namespace to communicate values. Although widely used, assigning a value to a location only suffices for one-to-one or one-to-many communication. Many-to-one communication results in data conflicts or collisions, when multiple values are stored in a location. Many data parallel languages leave the semantics of these conflicts undefined [14].

Other languages try to avoid potential errors due to data races by defining a semantics for these collisions [13, 22]. These languages typically use a binary reduction operator to combine colliding values into a single value that can be stored in a location. Reductions are also common in parallel applications, even those written in languages that do not provide first-class support for these operations. Unfortunately, in most data-parallel languages, this support consists of a limited set of predefined reduction functions—typically, the associative arithmetic and logical operations.

In this paper, we demonstrate that parallel languages need not and should not arbitrarily limit reductions in this way. User-defined reductions permit a programmer to extend these operations by defining new reduction functions on both scalar and compound data types. User-defined reductions are a natural generalization that subsumes the limited reductions offered by most languages. They permit a programmer to extend reductions in two dimensions: by formulating new operations for combining values, such as building a list from colliding values, and by extending reductions to user-defined data types. In addition, user-defined reductions offer a natural way to express many message-passing optimizations, such as update protocols and bulk data transfer, in a shared-memory program.

Some languages include user-defined reductions, such as Connection Machine Lisp [23], Paralation Lisp [19], and Fortran D [7]. We are unaware of published descriptions of implementations on MIMD machines. This paper describes an efficient implementation for user-defined reductions and contains experiments that demonstrate that user-defined reductions both simplify writing parallel programs and improve the performance of programs written in a high-level parallel language. With user-defined reductions, our high-level parallel programs ran at comparable or faster speeds than hand-coded parallel programs.

We performed this work in the context of the data-parallel language C**, although the results are applicable to other data-parallel languages. C** is a data-parallel language that extends C++ with a parallel aggregate data type and data-parallel operations. Section 2 describes C**'s large-grain data parallel execution model. In an imperative language, such as C**, many-to-one communication operations arises from assignment statements, which C** generalizes to reduction assignments that allow a programmer to specify a combining function for collisions. Extending reduction assignments to allow user-defined functions requires only a small syntax change, but the semantics require more careful consideration. In particular, user-defined reduction functions differ from primitive arithmetic operations because they may have side-effects and need not be associative or commutative. In addition, a data-parallel language must specify if and when intermediate results of a reduction assignment are visible to the program. Section 3 explains our deferred reduction model, in which the results of a reduction is only visible at the end of the parallel phase.

C**'s basic implementation of reductions uses messages to send values to a home processor. The deferred reduction model enables the C** runtime system to combine messages to the same destination and perform reductions locally, which reduces the message overhead. These simple techniques apply equally well to predefined reductions and permit standard arithmetic reductions to be efficiently implemented in the same framework. The compiler also uses processor combining trees to implement reductions on a parallel operation's return values. Section 4 describes the implementation in more detail.

We evaluated our implementation by comparing C** versions of five benchmarks (one small and four medium-size applications) against well optimized versions of the programs on a 32 processor CM-5. All of

| Program | Compared against | C** | SPMD | (SPMD/C**) |
|---|---|---|---|---|
| DSMC | Hybrid SM-MP | 74.2 | 82.7 | 0.90 |
| EM3D | Hybrid SM-MP | 10.7 | 5.0 | 2.14 |
| Water | Shared memory (SM) | 13.0 | 13.6 | 0.96 |
| Moldyn | Hybrid SM-MP | 27.0 | 26.7 | 1.01 |
| FFT | Shared memory (SM) | 2.0 | 8.5 | 0.11 |

Table 1: Comparative benchmark execution times (in seconds on 32-processor CM-5)

```
class Grid(float) [][])  {    /* Member functions */     };
```

Figure 1: Aggregate definition syntax in C**

these programs' communication patterns were fully captured by reduction assignments (both user-defined and primitive). Table 1 summarizes our results. Section 5 describes the benchmarks in more detail and presents complete performance results.

## 2 User-defined Reductions in C**

Hillis and Steele popularized the data-parallel programming model for programming massively parallel processors [10]. The high-level features of this model, such as a global namespace and nearly deterministic execution, prompted the design of many other data-parallel programming languages. Our language, C** provides coarse-grain data-parallelism, but enforces independent execution of coarse-grain tasks. Section 2.1 briefly describes the data-parallel execution model in C**, a detailed description of C** can be found elsewhere [13]. Section 2.2 describes the syntax of reduction assignments in C** and shows how user-defined extensions can be added with minor syntax extensions.

### 2.1 Data-parallelism in C**

Data-parallel programs express parallelism by invoking a data-parallel operation on a data collection. An invocation simultaneously executes data-parallel tasks for each element of the collection. The compiler and run-time system map tasks to physical processors in a machine.

Data collections are specified using aggregate types (e.g., arrays in HPF [9], vectors in NESL [2], special class constructs in PC++ [14]). C** overloads the class definition mechanism of C++ to define data collections that are called Aggregates. For example, Figure 1 declares a two-dimensional collection of floating point values whose size is specified when `Grid` objects are created.

Data-parallel operations in C** are specified using the `parallel` keyword. A parallel function operates on its data collection argument, which is also denoted by a `parallel` keyword. For example, Figure 2

```
void stencil(parallel Grid &A) parallel
{
    A[#0][#1] = (A[#0 - 1][#1] + A[#0][#1 - 1] + A[#0 + 1][#1] + A[#0][#1 + 1]) / 4.0;
}
```

Figure 2: Stencil in C**

```
float g;

void sum(parallel Grid &A) parallel
{
    g =%+ A[#0][#1];
}
```

Figure 3: Sum reduction assignment

describes the stencil operation in C**. The pseudo variables #0 and #1 identify row and column positions within the collection and allow access to neighboring elements.

C** provides *large-grain* data-parallelism [13]. It allows coarse-grain parallel tasks, but enforces task independence by providing local copies for global updates (i.e., copy-in, copy-out semantics). In the stencil function (Figure 2), each point in the grid receives the average of the old values of neighboring elements.

## 2.2 Reduction Assignment Syntax

*Reduction assignments* augment assignment statements with a combining or reduction operator. When an assignment statement executes, the statement's combining operator reduces the value in the location specified by the left-hand-side and the value from the right-hand-side into a new value, which is stored in the target location. For example, Figure 3 uses a reduction assignment to sum the values of a data collection.

In C**, *user-defined reduction assignments* required a minor syntax extension to allow function names as reduction operations. For example, Figure 4 shows a location reduction from the C** implementation of Dijkstra's algorithm to compute the shortest paths from a single source to all other nodes in a graph. Given an array of numbers, the location reduction identifies the minimum value in the collection, along with its position in the array. In Dijkstra's algorithm, this step finds the next node with the shortest path. The `min` parallel function pairs the node's distance and its position and applies the user-defined `loc_reduce` combining function to compute the location and the minimum value.

Note that the reduction function `loc_reduce` is not a symmetric binary operator with type `Loc` $\times$ `Loc` $\rightarrow$ `Loc`. The current C** compiler requires that the first parameter of a user-defined reduction serve as both input and output (following C syntax, it is a pointer).

# 3 Semantics of User-Defined Reductions

This section explores four language design issues associated with user-defined reductions. Two issues arise from the increased generality of the user-defined functions interfering with the well-defined behavior guarantees of a data-parallel language. The third issue identifies program points at which the results of a reduction can be made available. The fourth issue generalizes reductions into two separate components.

## 3.1 Semantic Problems

In most data parallel languages, the built-in, primitive reduction functions are side-effect free and cause no data access conflicts. However, user-defined reduction functions do not share this property. Conflicts between user-defined reduction functions can undermine a data-parallel language's semantic guarantees.

```
/* User-defined data type for Loc reduction */
struct Loc {
   int distance, node;
};

/* User-defined location reduction function */
void loc_reduce(Loc *lhs, Loc rhs)
{
    if (lhs->distance > rhs.distance)
        (*lhs) = rhs;
}

/* Aggregate storing the shortest paths */
struct Dijkstra (int) [N] { ...  };

/* Find the next node with the shortest path */
void min (parallel Dijkstra &d, Loc *result) parallel
{
    struct Loc temp;

    temp.node = #0;
    temp.distance = d.distance;
    *result =%loc_reduce temp;
}
```

Figure 4: Location reduction in Dijkstra's algorithm

We see three possible approaches to preserving a language's semantics, such as C**'s guarantee of nearly deterministic execution. First, user-defined reductions can be restricted to use only well-behaved functions that a compiler ensures are side-effect free. Although plausible, this rule is too restrictive because of limited compiler analyses in languages that support pointers and aliasing. Second, a language may permit compiler directives, such as HPF's INTENT directive, that assert properties of user-defined reductions that a compiler is unable to prove. This approach opens the door to difficult-to-find errors if a directive is incorrect. Third, a language may allow general functions, but require a run-time system to identify data access conflicts, as in Steele's Parallel Scheme [22]. Run-time conflict identification can be expensive and complex. The C** compiler relies on user guarantees that user-defined functions are safe.

Another issue is that user-defined reductions may not be commutative or associative, so that different combining orders lead to non-deterministic results. This is not a problem for two reasons. First, user-defined reductions are typically effectively associative [19] functions in which the absence of associativity does not affect a program's result. For example, the combining function `append` collects values into a list. In many cases, the list is a set so that the order of elements is unimportant. Second, a programmer can force a specific combining order by collecting all values into a list, sort them, and then combining. Mandating a combining order for all reductions unnecessarily restricts language implementors and imposes overhead on applications that do not require it.

## 3.2 Reduction result availability

In coarse-grain data-parallel languages, a task may continue after executing a reduction assignment. If a variable `target` is the target of a reduction assignment, the language must specify the value seen by subsequent accesses to `target` by the task. Three approaches are possible:

1. The language may prohibit accesses to `target`, except as a reduction target, as does Fortran D [7]. Erroneous accesses can be identified syntactically. This approach allows the runtime system to defer updating the target. However, syntactic analysis may not identify all erroneous accesses, particularly those involving arrays or pointers.

2. The language may retain the old value of `target` after a reduction. When the data-parallel operation completes, the colliding values can be combined and used to update reduction targets. We call this approach *deferred reductions.*

3. The language may defer combining, but update the local copy of `target` by merging contribution from the local task. This approach is suitable for a language like C**, which mandates local copies to enforce independence. However, other data-parallel languages do not make such a clear distinction between local and global values and are better off using deferred reductions.

## 3.3 User-Defined Updates

A reduction assignment is actually two actions, combining conflicting values and updating the target location with the new value. For example, in C**, the reduction assignment `g =%+ A[#0][#1];` from Figure 3 adds the sum of elements of `A` to `g`. Suppose that each element of `A` is a 3-element array of floating point values (representing a force vector). To sum these vectors, the programmer can use an `array_sum` function thus: `g =%array_sum A[#0][#1];`.

A user-defined update is syntactic sugar for updates to a target location of a different type than the combined value. To continue the example, if `g` is not a vector but a list of vectors representing sums of `A` at different points in the program, the programmer can append the current sum the list with `g =%append : array_sum A[#0][#1];`. Without user-defined updates, the programmer must store the sum in a temporary and append to `g` later. To prevent non-deterministic results, update functions must follow the same restrictions as reductions (Section 3.1).

C** also permits a programmer to omit the combining function entirely and specify only an update function. In this case, the update function is invoked on each value separately. For example, to collect all reduced values in a list, the programmer provides an update function that inserts a single element into the list. This approach is also useful when combining values is more expensive than merging values one-by-one, as when each value modifies distinct parts of a large data structure [26].

## 4 Implementation

The C** compiler implements user-defined reductions with a small amount of runtime support. This section describes how the compiler and runtime system implement basic and update reductions (Section 4.1), exploit the deferred reduction model to vectorize messages (Section 4.2), and combine values locally to reduce message traffic (Section 4.3). In some cases, the compiler also uses a combining tree to reduce values (Section 4.4).

## 4.1 Basic and Update Reductions

A reduction assignment updates its target with the combined value of colliding right-hand-side values. The C** implementation involves two processors: the processor that executes the reduction assignment (processor A) and the processor that owns the target location (processor B). Processor A, which executes the reduction, sends processor B a message containing three items: the right-hand-side value, the combining function descriptor, and the target location pointer. At the end of the parallel phase, Processor B collects incoming reduction messages, combines colliding values and updates target locations. To implement update reductions (Section 3.3), processor B replaces the combining function with an update function.

The "owner-updates" model is simple to implement and requires minimal runtime system support. It depends on the runtime system to support target location queries, which is usually available in languages that provide a global name space.

## 4.2 Bulk reductions

During a data-parallel operation, a processor may execute multiple reductions for two reasons. First, the number of data-parallel tasks is usually much larger than the number of processors, so each processor runs multiple tasks. Second, each coarse-grain data-parallel task may execute multiple reduction assignments. The deferred reduction model allows the compiler to defer sending reduction messages until the end of the parallel phase. This permits several messages to the same destination processor to be bundled into a single message, which is typically far more efficient to send and receive.

This optimization is essential when the application program uses reductions to communicate large amounts of data. As the graphs in Section 5 show, this optimization improved program performance between 1.06x (Tiled FFT) and 6.76x (EM3D).

## 4.3 Local Combining

If a processor executes multiple reductions to the same target, the values can be combined locally before being sent for global combining. Local combining requires the runtime system to identify common targets locally, for which the C** system uses a hash table of target addresses. Probing this table increases the overhead of the reduction, but allows for a decrease in communications cost. This is a good example of an optimization that trades off worse sequential performance for better communication (and therefore parallel) performance.

On Water and Moldyn (two benchmarks that benefited from local combining), (Section 5), this optimization improved performance by 1.21x and 2.32x respectively. Figure 6 also demonstrates the tradeoff between sequential and parallel performance for Water. The Splash version is consistently faster (as much as 1.6x on one processor) up to 16 processors, but the C** version is 1.06x faster on 32 processors.

## 4.4 Tree Combining

C** also allows reductions to combine values returned by parallel tasks [13]. The combined return value is returned to the sequential phase as the result of a data-parallel operation. C** executes return reductions using processor combining trees for efficiency, much like combining tree barriers for synchronization [15]. The processors are organized as a tree. Results move up the tree, bounding the communication latency by $O(logP)$, where $P$ is the total number of processors.

| Program | Scientific Domain | Input data set | SPMD size | C** size |
|---|---|---|---|---|
| DSMC | Particle-in-cell | 9720 cells, 400 iterations initially 48600 particles finally 72500 particles | 5,000 | 2,258 |
| EM3D | Electromagnetics | 32000 nodes, 20% remote edges degree 5, 100 iterations | 3,175 | 254 |
| Water | Molecular dynamics | 512 molecules, 30 iterations | 2,278 | 1,231 |
| Moldyn | Molecular dynamics | 16384 molecules, 30 iterations | 3,226 | 758 |
| FFT | Signal processing | 262144 complex numbers | 964 | 499 |

Table 2: High-level benchmark description and program sizes (in lines)

## 5 Performance

This section evaluates the performance of C** optimizations on five benchmark programs and compares them against highly optimized alternatives written in SPMD style. Our C** compiler targets a Thinking Machines CM-5 [11], a message-passing multiprocessor. The run-time system builds on a portable parallel substrate called Tempest [17], which provides mechanisms for shared memory and message passing on a wide range of parallel machines. The compiler uses the Tempest shared-memory mechanisms to implement a global namespace and its message-passing primitives to implement reductions. Table 2 gives a high-level description of the benchmarks, including the input data sets.

For three applications, DSMC, Moldyn, and EM3D, we compared against the best optimized programs previously written (by others) using a hybrid of shared-memory and message-passing techniques [5, 16]. These programs use transparent shared memory as a basis, but communicate select data structures through custom shared-memory or message-passing protocols. For three programs, including DSMC and Moldyn, Mukherjee et al. demonstrated that this approach compares favorably with the well-known Maryland CHAOS library [4] for irregular applications. Falsafi et al. showed that the best optimized version of EM3D ran faster than a comparable message-passing program [5].

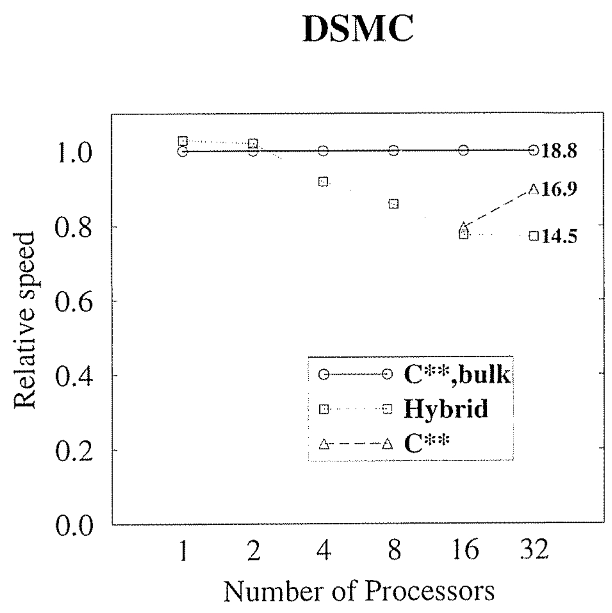
The other two applications (Water and FFT) are transparent shared memory programs from the Stanford SPLASH suite [24]. We ran these programs on Blizzard, a fine-grain distributed shared memory system that runs on the CM-5 [20].

Table 2 also compares the program size (number of lines) of the C** version of each benchmark against the optimized versions. In all cases, the C** programs are significantly smaller. The difference is more pronounced for the Hybrid codes (e.g., EM3D) that contain custom code to improve communication performance.

### 5.1 DSMC

DSMC simulates particle movement and collision in a three dimensional domain using a Discrete Simulation Monte Carlo method [21]. DSMC divides the domain into cells in a static Cartesian grid and distributes molecules among cells. At each time step, the algorithm moves molecules under the influence of forces from interactions with other molecules, adds new molecules from a jet stream, and collides molecules in the same cell. The data-parallel implementation of DSMC exploits parallelism on cells.

The move phase updates the positions of each cell's molecules from their current velocities. In the process, molecules may move from one cell to another (usually neighboring) cell. Inter-cell molecule transfer is many-to-many communication since a cell may export molecules to different destinations and a cell may receive multiple molecules. The C** implementation uses an update reduction function to collect entering



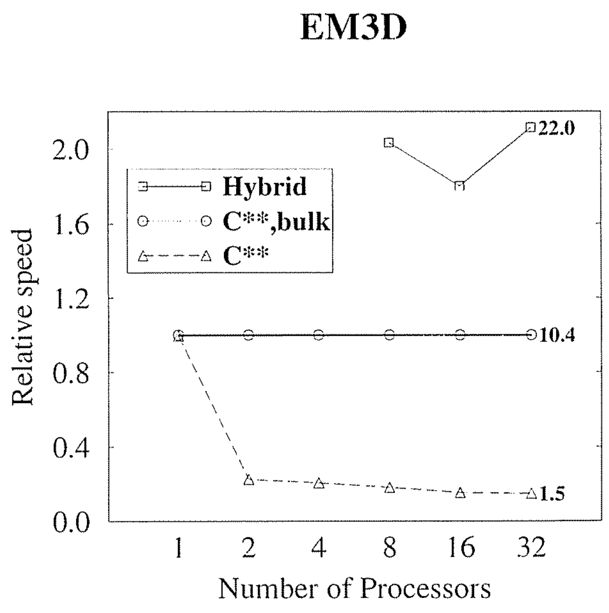


Figure 5: Execution speeds of 3 versions of DSMC and EM3D, normalized to the optimized C** version (with bulk reductions). Speeds greater than 1.0 are faster than the C** version. The numbers adjoining the curves are speedups relative to a sequential version.

molecules and add them to the cell's list. The Hybrid implementation uses bulk messages for molecules moving to neighboring processors and single messages for other molecules.

Figure 5 compares the relative speeds of the Hybrid version against C**'s unoptimized and bulk reduction versions. The bulk reduction improves significantly over the simple reduction (1.3x) and is 1.11x faster than the optimized Hybrid version.

## 5.2 EM3D

EM3D models the propagation of electromagnetic waves through objects in three dimensions [3]. The problem is formulated a bipartite graph of H nodes representing magnetic fields and E nodes representing electric fields, with directed edges between H nodes and E nodes. Each time step consists of two parts: first, each H node accumulates the effects of neighboring E nodes, and then each E node accumulates the effects of neighboring H nodes.

In C**, the effects are accumulated with addition reductions. In the first part of a time step, E nodes send their values (using reductions) to H nodes, where they are collected and combined. The defered reduction implementation mimics the producer-consumer data movement pattern of the program, which is essential to good performance [5]. The Hybrid version of EM3D uses a custom update protocol to transfer data in the producer-consumer pattern.

Figures 5 compares the relative speeds of the Hybrid version with the simple and bulk-reduction C** versions. The bulk-reduction optimization improves the execution time by 6.76x. However, the Hybrid version is still 2.11x faster than the best C** version. The primary reason for this is the overhead of collecting reductions into bulk messages (and testing for buffer overflow) compared to the computation itself (one double-precision addition). This difference is magnified on a single processor: a uniprocessor version of the C** code was 3x slower than its sequential counterpart. The Hybrid version also exploits the repetitive

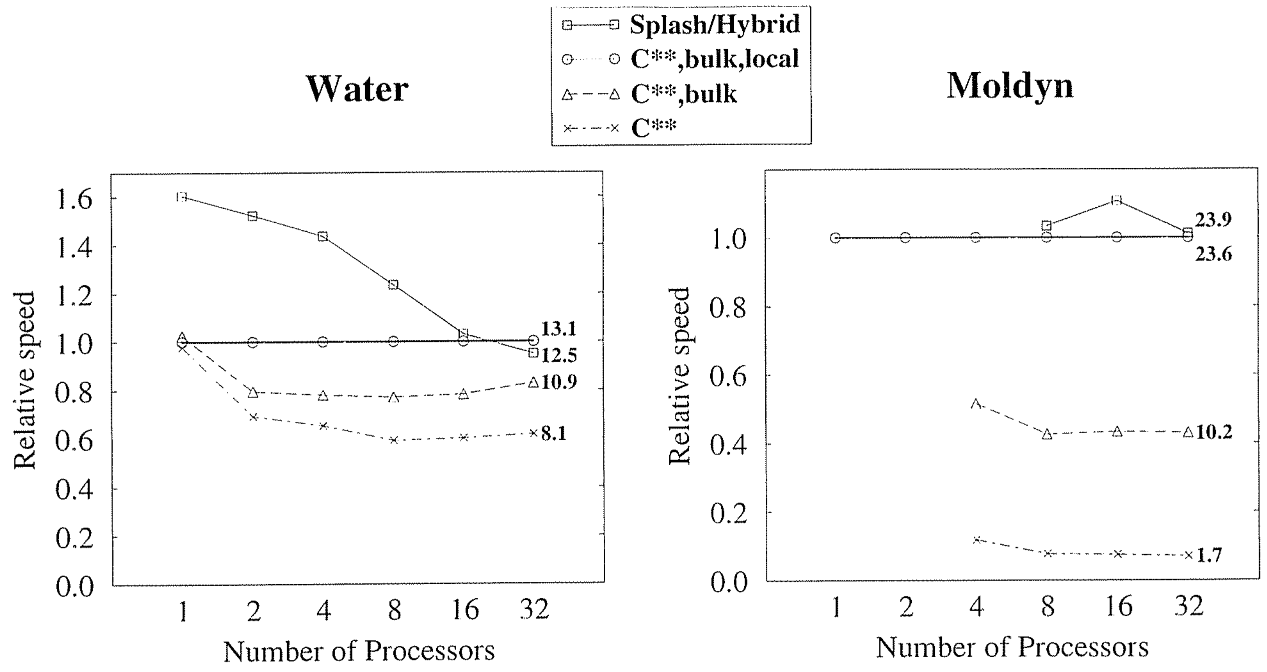


Figure 6: Execution speeds of 4 versions of Water and Moldyn's force computation phase relative to the optimized C** version (with bulk reductions and local combining). Speeds greater than 1.0 are faster than the C** version. The numbers adjoining the curves are speedups relative to a sequential version.

communication pattern to utilize message-passing channels which are faster than bulk messages on the CM-5. This difference accounts for 17.5% of C**'s execution time slowdown compared to Hybrid.

## 5.3 Water and Moldyn

Water and Moldyn are well-known molecular dynamics code used to model macromolecular systems [16]. Molecules are initially distributed uniformly in a cuboidal region with a Maxwellian distribution of initial velocities. A molecule moves under the influence of forces exerted on it by other molecules. In Moldyn, the force computation limits interactions to molecules within a cut-off radius by maintaining an interaction list that is updated infrequently. Water [24] computes interactions between all pairs of molecules.

Evaluating an interaction involves reading the positions of two molecules, computing the resulting force, and updating each molecule with the resultant force. Force update involves many-to-one communication; each molecule receives force increments from many interacting molecules. In C**, the force increments are combined using a reduction function (with local combining). The Hybrid implementation of Moldyn stores local copies of force increments for all molecules on each processor. The local copies are combined in an efficient ring reduction using messages. These local copies add significantly to the memory requirements, especially when the number of processors is large and each processor is responsible for a small fraction of the molecules. By contrast, C** allocates space only for the reductions actually executed on a processor, which saves significant amounts of memory. Water also uses local copies, but uses locks to synchronize combining.

Figure 6 compares the relative speeds of the Hybrid (and Splash respectively) version of Water and Moldyn's force computation phase with C** versions at three levels of optimization (simple reductions, bulk reductions and bulk reductions with local combining). We compared only the force computation phase in Moldyn because the C** version used a faster algorithm for building the interaction list.

In Moldyn, bulk reduction provides a 6.04x improvement in execution time and local combining improves

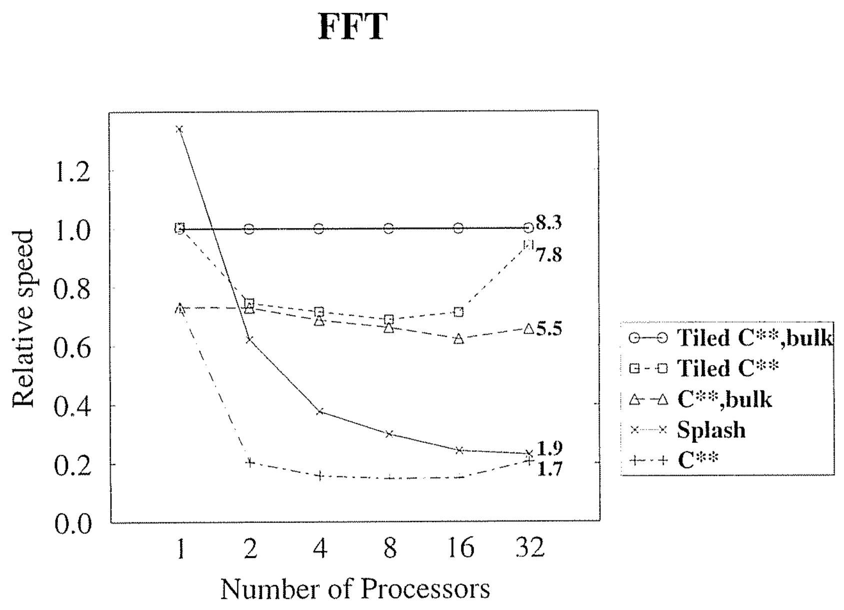


Figure 7: Execution speeds of 5 versions of FFT relative to the optimized Tiled C** version with bulk reductions. Speeds greater than 1.0 are faster than the C** version. The numbers adjoining the curves are speedups relative to a sequential version.

further by 2.32x. On 32 processors, the highly optimized Hybrid implementation is only 1% faster than the best C** version.

Water simulates fewer molecules, and, as a result, communicates less data. The improvements from bulk optimization (1.34x) and local combining (1.21x) are impressive, but not as large as in Moldyn.

The sequential overhead of local combining is demonstrated by Water. Between 1 and 16 processors, the sequential overhead dominates, but at 32 processors, the lower cost of communication overshadows the sequential overhead. As a result, the best C** version is 1.05x faster than the Splash version.

## 5.4 FFT

The FFT kernel, from the SPLASH suite [24], implements a complex 1-D version of the radix-$\sqrt{N}$ six-step algorithm that minimizes interprocessor communication [1]. The input 1-D vector of size $N$ is organized as a $\sqrt{N} \times \sqrt{N}$ square matrix. The most expensive phases of the algorithm are the three matrix transpose phases, each of which involve all-to-all interprocessor communication.

We compared two versions of the C** code against the SPLASH version running on Blizzard. The first C** version uses null reductions to move each point in the matrix to its transposed position. The second C** version partitions the matrix into $4 \times 4$ tiles at the source level to increase cache reuse [25]. The user-defined reduction function transfers a tile at a time and transposes the tile at the receiving end in an update function.

Figure 7 compares the relative speeds of the Hybrid version with the simple and bulk-reduction versions of both C** programs. The tiled C** version is 1.3x faster than the simple C** version because of cache reuse and because the larger tiles amortize the reduction overhead (target pointer, combining function descriptor) over larger data blocks. The bulk reduction optimization improves the speed of the simple and tiled C** programs by 3.94x and 1.33x respectively.

The best C** version is also significantly faster than the Splash shared-memory version, but this difference is due in a large part to the high overhead of the Blizzard shared-memory implementation.

## 6 Related work

Many previous papers have recognized the need for powerful reduction operators. For example, Dataparallel C adds a tournament operator [8] to carry along an extra value in max-reductions. Also, in comparing the message-passing and data-parallel paradigms, Klaiber et al. [12] noted the inefficiency in expressing many-to-many communication in C*. To remedy the problem, they introduced the `sendToQueue` reduction operator that is similar to `append`. Sharma et al. [21] used the intrinsic `list` operator to efficiently execute the particle-in-cell application DSMC. Several applications in the HPF-2 motivating applications suite [6] note their requirement for user-defined reductions.

This paper describes a general mechanism supporting powerful reduction operators. Reductions are a feature in most, if not all, data-parallel languages. A few of these languages allow user-defined functions for reduction operations (e.g., Connection Machine Lisp [23], Paralation Lisp [19], and Fortran D [7]). However, we are unaware of papers describing implementations of user-defined reductions on parallel machines.

## 7 Conclusion

Data-parallel languages mitigate the difficulty of parallel programming by providing high-level abstractions for concurrency, synchronization and communication. In data-parallel languages, reductions express many-to-one communication patterns by combining multiple colliding values using a binary reduction operator. This paper demonstrates that user-defined reductions are a useful addition to a data-parallel language in two respects: they allow the programmer to express powerful combining operators, and they can be implemented efficiently with minimal runtime support.

User-defined reductions are useful because they generalize reductions in two dimensions. First, they support powerful combining operations (e.g., location reductions and list building) in a familiar framework. Second, they generalize reductions to user-defined data types. Unconstrained user-defined reductions, however, can violate a language's safety guarantees. In this paper, we identify language design issues that a language designer must consider, and explore possible options for these issues.

We also describe the implementation of user-defined reductions in the data-parallel language C**. The basic implementation uses messages to communicate reduction data. Two simple and well-known optimizations — message vectorization and local combining — significantly improve the execution speed of applications using reductions.

C** versions of 5 benchmarks using reductions performed well compared to equivalent, optimized hand-coded SPMD programs running on a 32-processor CM-5. In the worse case, the C** version was 2x slower than a message-passing code. It was between 1% slower and 10% faster on 3 other codes, and 4.25x faster on a communication intensive shared memory program. Given the complexity and effort in tuning the SPMD codes, the C** programs are far more attractive.

## References

[1] David H. Bailey. FFTs in External or Hierarchical Memory. *The Journal of Supercomputing*, 4(1):20–??, March 1990.

[2] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie Mellon University, April 1993.

[3] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[4] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[5] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[6] High Performance Fortran Forum. HPF-2 Scope of Activities and Motivating Applications, November 1994. Available at ftp://hpsl.cs.umd.edu/pub/hpf_bench/hpf2.ps.

[7] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR900749, Centre for Research on Parallel Computation, Rice University, December 1990.

[8] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0, May 1993.

[10] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[11] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[12] Alexander C. Klaiber and James L. Frankel. Comparing Data-Parallel and Message-Passing Paradigms. In *Proceedings of the International Conference on Parallel Processing*, pages II-11–II-20, August 1993.

[13] James R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages And Compilers for Parallel Computing (5th International Workshop)*, pages 326–341. Springer-Verlag, August 1993.

[14] Jenq Kuen Lee and Dennis Gannon. Object Oriented Parallel Programming, Experiments and Results. In *Proceedings of Supercomputing '91*, pages 273–282, Albuquerque, NM, November 1991.

[15] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[16] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[17] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[18] John R. Rose and Guy L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing*, pages 2–16, Santa Clara, California, May 1987.

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

[20] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[21] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proceedings of Supercomputing '94*, pages 97–106, November 1994.

[22] Guy L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 218–231, January 1990.

[23] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine LISP: Fine-Grained Parallel Symbolic Processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 279–297, August 1986.

[24] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[25] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 219–229, San Jose, California, 1994.

[26] Bwolen Yang, Jon Webb, James M. Stichnoth, David R. O'Halloran, and Thomas Gross. Do&Merge: Integrating Parallel Loops and Reductions. In Utpal Bannerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (Proceedings of the Sixth Internationa Workshop)*, pages 169–183, Portland, Oregon, August 1993. Springer-Verlag.