HIGH-LEVEL LANGUAGE SUPPORT FOR USER-DEFINED REDUCTIONS

Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder

University of Washington, Seattle, WA 98195-2350 USA
{deitz,brad,snyder}@cs.washington.edu

Abstract. The optimized handling of reductions on parallel supercomputers or clusters of workstations is critical to high performance because reductions are common in scienti£c codes and a potential source of bottlenecks. Yet in many high-level languages, a mechanism for writing ef£cient reductions remains surprisingly absent. Further, when such mechanisms do exist, they often do not provide the ¤exibility a programmer needs to achieve a desirable level of performance. In this paper, we present a new language construct for arbitrary reductions that lets a programmer achieve a level of performance equal to that achievable with the highly ¤exible, but low-level combination of Fortran and MPI. We have implemented this construct in the ZPL language and evaluate it in the context of the initialization of the NAS MG benchmark. We show a 45 times speedup over the same code written in ZPL without this construct. In addition, performance on a large number of processors surpasses that achieved in the NAS implementation showing that our mechanism provides programmers with the needed ¤exibility.

1 Introduction

Reductions require careful compilation for two reasons. First, they abound in scienti£c codes. For example, they are used in algorithms for image processing and for computational geometry, in kernels for matrix multiplication and for sorting, and to test for convergence in iterative algorithms. Often a test for convergence is repeated at frequent intervals throughout the program. Second, performance, often whole program performance, suffers if a reduction is poorly optimized or left unoptimized. An unoptimized reduction is sequential or a source of signi£cant unnecessary communication. Reductions are non-trivial to parallelize; there are dependences between loop iterations that can only be violated if it is known that an operation or function is associative.

A reduction is a mapping from an array of n dimensions to an array of less than n dimensions or a scalar. In this mapping, collisions (two or more array elements mapping to the same part of the result) must occur because the result is necessarily smaller than the original array. To resolve these collisions, elements mapping to the same location are combined with an operator that is almost always binary. Moreover, this binary operator is usually associative and commutative. If the operator is associative, the parallel-pre£x method [11] can be used to parallelize the reduction. If the operator is commutative, the computation might be further optimizeable on some parallel computers. It is essential to take advantage of associativity when compiling for a parallel computer. Otherwise potential parallelism is left unexploited.

As examples of reductions, consider the following two. In a full sum reduction over an array of integers, the operator is addition and the result is the sum of every integer in the array. In a partial sum reduction (often called a histogram reduction) over a 2D array of integers the operator again is addition but the result is either a column of integers where each is the sum of all the integers in its row or a row of integers where each is the sum of all the integers in its column. In addition to summation, other common reductions include the following: determining the minimum or maximum value in an array, £nding the location of the minimum or maximum value in an array, and calculating the logical or bit-wise “and” or “or” of the elements in an array.

In this paper, we introduce a parallel language construct that lets a programmer explicitly specify arbitrary reductions so that the compiler cannot fail to exploit associativity by parallelizing the reduction. We report on our implementation of this construct in the context of the ZPL language [18]. Most high-level parallel languages include reductions in their repertoire of devices, but fewer allow for the speci£cation of arbitrary reductions. Languages like NESL [3] (and ZPL prior to our introduction of this mechanism) supply a number of built-in reductions such as those mentioned above, but do not let programmers de£ne their own. Though languages like C** [12, 13] and SAC [16] have user-de£ned reductions, the mechanisms do not allow for reductions to be written as ef£ciently as with the mechanism we introduce.

The rest of this paper is organized as follows. In Section 2 we consider the trade-offs between various programming approaches to reductions. In Section 3 we introduce the ZPL language and in Section 4 we present our mechanism to ef£ciently support arbitrary user-de£ned reductions in the context of ZPL. In Section 5 we quantitatively evaluate this mechanism. In Sections 6 and 7 we discuss related work and conclude.

2 Programming Approaches to Reductions

This section concerns itself with the various approaches one takes to program parallel supercomputers or clusters of workstations. In particular, we focus on how easy it is to write a reduction using a given approach and what level of performance is likely to be achieved. The approaches we consider are as follows: using an automatic parallelizer, relying on a parallelizing compiler assisted by programmer-inserted directives, employing a message passing library, or writing code in a high-level language.

2.1 Automatic and semi-automatic parallelization

Most research related to improving state-of-the-art programming practices with regard to reductions lies under the umbrella of automatic parallelization. With this approach, programming is as easy as writing code for a single processor because this is just what the programmer does. The compiler is solely responsible for exploiting parallelism. Traditionally pattern matching and idiom recognition have been used to parallelize reductions [4, 14]. Sophisticated techniques for recognizing broader classes of reductions have also been examined [8, 19]. Commutativity analysis [15] promises to be yet another effective technique.

However, it is an undecidable problem to determine whether a function is associative [10]. Moreover, even if a function is not technically associative, the salient part of the calculation might be. To exploit the associativity of a function it is suf£cient but not necessary that the function be associative. Automatic parallelization is an invaluable technique for quickly improving performance on large legacy codes written for sequential processors, but one cannot expect to achieve consistently high performance. There are too many uses for different reductions that no compiler will ever be able to identify and parallelize all reductions. Moreover, the compiler might justi£ably determine a reduction is not parallelizeable even if the programmer is able to determine that it is.

It is not a new observation that a compiler needs assistance in parallelizing codes. This observation lead to the development of High Performance Fortran (HPF) [9]. HPF limits the risk that a critical section of code is left unparallelized by relying on the programmer to insert directives into a code. A programmer writes a sequential program in ordinary Fortran and then adds data layout directives thereby creating an HPF program. Programmers of HPF have achieved successes but still suffer from many of the same problems encountered by programmers relying on fully automatic parallelization. Compilers are, after all, supposed to recognize reductions and parallelize them accordingly.

2.2 Message passing libraries

Message passing libraries shift the responsibility of exploiting parallelism to the programmer. Details of communication account for a large portion of the code [6] and valuable time must be spent writing it. In addition, the programmer must write the computation based on a per-processor view of the system. The negative impact this has on code readability and maintainability should not be underestimated. In spite of these problems, this approach to programming parallel computers is the standard method employed by scientists who demand high performance.

Reductions are not dif£cult to write using a message passing library. MPI [17] comes with a rich set of built-in reductions, but occasionally the reduction a programmer wants to write is not in this set. In this case, MPI has a mechanism that allows for user-de£ned reductions. The programmer must write a function that can be used by processors to combine their own local results. This function is associated with a datatype and can be used in the standard reduction function call. For example, see the code in Appendix A in which it is assumed that a reduction to return the minimum element in an array as well as its location is absent from the built-in set of reductions in the MPI library. The disadvantage to using message passing libraries is that to do so means programming with a low-level per-processor view of the system. Even though this simpli£es what needs to be added to support user-de£ned reductions, the overall complexity level is higher than the next approach we consider.

2.3 High-level parallel languages

High-level parallel languages, like message passing libraries, are used by programmers who want a guarantee of parallelization. Reductions written in these high level languages are guaranteed to be parallel. The advantage over message passing libraries is two-fold; details of communication are hidden from the programmer and the view of the computation is global, i.e., not on a per-processor level. The disadvantage is that a certain level of control is lost. In languages that do not provide user-de£ned reductions, like NESL (and ZPL before the mechanism in this paper was implemented), a programmer often must write grossly ineff£cient code that relies on some simple, built-in reductions to accomplish what could easily be done with a more complicated reduction.

For example, suppose a programmer wants to determine the two smallest elements in a large array. Ideally, each processor would compute the two smallest elements on the part of the array owned by that processor. Then a reduction could be computed in which pairs of processors compare four elements, the two smallest elements on each processor, to determine the smallest two elements on both of the processors. This is an associative operation and so the parallel-pre£x method can be used to reduce the two smallest elements on all the processors in parallel.

If a high-level language only supplies a few built-in reductions including one in which the single smallest element in an array as well as its location in the array can be identi£ed, the most ef£cient solution is unworkable. Instead, we must £nd the smallest element in the array and replace it with an element of maximum value. Then we can £nd the second smallest value using the same reduction. Finally we should replace the smallest value if the array is to be left in its original state. Clearly this is ineff£cient and motivates the need for a powerful language construct that allows for arbitrary, user-de£ned reductions. Note that in the case of ZPL before this mechanism was implemented, the best reduction we could use to £nd the two smallest elements was a reduction that determined the smallest element in the array, but not its location. So an even less ef£cient algorithm for £nding the smallest two elements in an array would have to be used.

In this paper, we improve upon user-de£ned reduction mechanisms previously proposed for high-level, parallel languages. The language construct we add to the ZPL language lets a programmer achieve the performance hitherto only achievable with message passing libraries.

3 A Brief Introduction to ZPL

ZPL is a high-level, data-parallel, array-based language used to program parallel computers when high performance is desired even if development time is limited. Its relative simplicity and Pascal-like feel make it easy to read and understand, yet it also retains a sophisticated model of parallelism. For these reasons and because we have implemented user-de£ned reductions in ZPL, we choose to introduce our language construct in the context of ZPL. It should be noted however that this construct is suf£ciently general to apply to other high-level parallel languages. In this section we introduce those features of the ZPL language relevant to this paper. Interested readers are referred to the user's guide [18].

3.1 Regions and arrays

Central to ZPL is the region [7]. Regions are index sets with no associated data. To declare two regions, `R` and `BigR`, such that `BigR` is an $n+2 \times n+2$ index set and `R` is an $n \times n$ index set that refers only to the non-border portion of `BigR`, we write the following:

```
region BigR = [0..n+1, 0..n+1];
       R    = [1..n, 1..n];
```

Regions are used in two contexts. First, they are used to declare parallel arrays. All arrays declared over regions are parallel and as such are distributed over the processors in a manner speci£able at runtime. We declare three integer arrays, `A`, `B`, and `C`, to be over the index set given by `BigR` by writing:

```
var A, B, C : [BigR] integer;
```

Non-parallel arrays, also called indexed arrays, are declared using the keyword “array”. These arrays are replicated on each processor and are guaranteed to contain the same data on each processor. To declare a ten element array of integers, a, that is replicated and consistent on all processors, we write:

```
var a :  array[1..10] of integer;
```

Note that there is no region associated with this declaration. The index set is instead speci£ed after the array keyword. A second use of regions is to implicitly signal parallel computation. For example, to sum corresponding values in the non-border portion of parallel arrays A and B and store the result in C, we write the following line of code:

```
[R] C := A + B;
```

This corresponds to a doubly nested loop over the $n \times n$ index set. There is no communication since all interacting parallel arrays are distributed in the same way.

3.2 Parallel operators and communication

Communication only arises when certain ZPL operators are used. Since communication is a major overhead in parallel computing, programmers should avoid these operators whenever possible. Additionally, the operators that correspond to less communication should be used instead of operators that correspond to more communication. The classi£cation of these operators gives ZPL a performance model that lets programmers determine how fast or slow are their algorithms [5].

The most basic operator is the @ operator. This operator allows the programmer to refer to elements offset from elements in the array being assigned. It is important to note that this implies the possibility of communication, in particular, point-to-point or nearest-neighbor communication. To write a computation in which the sum of four adjacent elements in the array A are assigned to elements in the same array, we write the following code: (Note that because this is a data-parallel computation signaled by the region, only old values in the array A, values in A before any computation in this line of code occur, are used to update the array.)

```
[R] A := A@[0,1] + A@[1, 0] + A@[0,-1] + A@[-1,0];
```

A more expensive communication operation is the reduce operator, <<. This entails broadcast and/or parallel-pre£x communication. There are a number of built-in reductions in the ZPL language such as summation, minimum value, maximum value, etc. For example, to calculate the sum of every value in the array A and store it in the £rst element of the array a, we write the following code:

```
[BigR] a[1] := +<< A;
```

The indexed array, a, is directly indexed into. The region does not apply to it, applying instead to the computation over A. Parallel arrays cannot be indexed into since this would allow for arbitrary communication patterns. Only indexed arrays can be indexed hence their name. Arbitrary indexing of parallel arrays can be done in bulk using the permute operator, #. This is the most expensive communication operator in the ZPL language.

4 A Mechanism for User-de£ned Reductions

We have added the ability to write user-de£ned reductions in the ZPL language by overloading functions. Each of the overloaded functions corresponds to a different piece of the reduction. In this section, we describe how this works. First, we present a simple example of using the user-de£ned reduction mechanism to de£ne one of the simple built-in reductions. We then describe two more complicated reductions that can be implemented in this way and that illustrate every aspect of our mechanism. Finally, we discuss a few miscellaneous issues related to user-de£ned reductions: associativity, commutativity, and aggregation.

4.1 Basic user-de£ned reductions

User-de£ned reductions in ZPL are easy to write. By overloading a function, the same reduction operator can be used. This ensures that the performance model remains intact [5]. As an example, suppose ZPL's built-in reductions did not include the sum reduction. Then we could realize the same computation by writing the code appearing in Figure 1.

```
1    procedure sum() :  integer;
2    begin
3       return 0;
4    end;
5
6    procedure sum(a, b :  integer) :  integer;
7    begin
8       return a + b;
9    end;

                 ⋮

10   ...  sum<< A ...;
```

Figure 1: User-de£ned `sum` reduction in ZPL

We have overloaded the function `sum`. The de£nition of `sum` in lines 1-4 of Figure 1 is the initialization function; in lines 6-9, the reduction function. For a full reduction, £nding the sum of every integer in an array, the parallel implementation works in the following way. Each processor has a single variable of type integer used to accumulate its local sum. On each processor, this variable is initialized with the initialization function. Then, on each processor, the variable is repeatedly assigned the result of the reduction function as it is applied to every element in the array residing on the local processor and the accumulating variable. Finally, the same function is used to combine accumulating values between pairs of processors. Using the parallel-pre£x method, this last step can be done in a number of steps on the order of the logarithm of the number of processors.

4.2 Generalized user-de£ned reductions and the `minten` reduction

In general, three functions must be used to correspond to the three phases of a reduction: the initialization, the local reduction, and the global reduction. In the initialization phase, all local accumulating values are initialized. In the local phase, the local function is applied to the local accumulating value and values in the array residing on the local processor. In the global phase, the global function is applied to accumulating values on different processors. In many cases such as the `sum` reduction, the local and global phases can be described with a single function.

To illustrate why three distinct functions might be desirable, consider the `minten` reduction. In the `minten` reduction, we are given an array of values and must £nd the ten smallest values. In ZPL, we can write this reduction as in Figure 2. Note that for ef£ciency, we write the procedures so that the result is not returned, but instead overwrites a parameter. This technique also applies to the `sum` reduction but is less important in that case since the result type is small.

In general, to specify a user-de£ned reduction that takes an array of type A elements and returns a lower rank array of type B elements, overloaded functions of the following types must be constructed: $\phi \rightarrow B$, $A \times B \rightarrow B$, and $B \times B \rightarrow B$. For ef£ciency, the functions may be speci£ed in the form $B* \rightarrow \phi$, $A \times B* \rightarrow \phi$, and $B \times B* \rightarrow \phi$ as is done in the `minten` reduction. The symbol "ϕ" corresponds to either no argument or no result and we use a "$*$" to denote a parameter that is passed by reference.

The advantage of allowing the programmer to distinguish between local and global functions is two-fold. First, greater ef£ciency is achieved by not requiring an A type to be translated to a B type before the reduction. Second, if the global operation is more compute-intensive than the local function, it is better to use the faster local function. It is used more often during the course of a reduction assuming many array elements reside on each processor. We realize both of these advantages in the `minten` code. It would be inef£cient to translate each array element into an array of ten elements containing the array element and nine maximum values. Further, the global function of £nding the ten

```
type vecten = array[1..10] of double;

procedure minten(var maxvec :  vecten);
var i :  integer
begin
   for i := 1 to 10 do
      maxvec[i] := MAXDOUBLE;
   end;
end;

procedure minten(newval :  double; var bestvec :  vecten);
var i :  integer;
    tmpval :  double;
begin
   for i := 1 to 10 do
      if (newval < bestv[i]) then
         tmpval := bestv[i];
         bestv[i] := newval;
         newval := tmpval;
      end;
   end;
end;

procedure minten(var bestvec1 :  vector; var bestvec2 :  vector);
var i :  integer;
begin
   for i := 1 to 10 do
      minten(bestvec1[i], bestvec2);
   end;
end;

         ⋮

...  minten<< A ...;
```

Figure 2: User-de£ned `minten` reduction in ZPL

smallest values in two arrays of ten values each is more compute-intensive than checking to see if a single element is smaller than any element in an array of ten elements and, if so, replacing that element.

4.3 Extensions for the `minloc` reduction

In addition to letting the programmer write the two functions for ef£ciency, letting the programmer pass in extra parameters to the local function adds to the potential ef£ciency. For example, to write a `minloc` reduction using the mechanism for user-de£ned reductions as described to this point, we would need to translate the array values into a new type that includes its location. Alternatively, we can pass in some extra information to the local function. The `minloc` reduction is similar to the basic `minimum` reduction but along with the minimum value, the reduction returns the location of the minimum value in the array. Figure 3 contains an ef£cient `minloc` reduction in ZPL.

Note the use of the Index1 and Index2 "arrays". These are not arrays in that no storage is associated with them. However, they can be thought of as read-only arrays. In general, the variable Index*d* contains the value i in the ith position of the *d*th dimension for all i de£ned by the current region. These variables are used in the local portion of the reduction and are passed in to the local function speci£ed by the user. Ordinary overloaded function resolution techniques still apply. The £nal argument is for the accumulating value.

Appendix A contains code that implements the `minloc` reduction in C and MPI. ZPL allows the programmer the same level of expressibility because the three functions correspond exactly to the points in the low-level C+MPI code

```
type minxy = record
                d :  double;
                x, y :  integer;
             end;

procedure minloc(var max :  minxy);
begin
   max.d := MAXDOUBLE;
end;

procedure minloc(d :  double; x, y :  integer; var best :  minxy);
begin
   if d < best.d then
      best.d := d;
      best.x := x;
      best.y := y;
   end;
end;

procedure minloc(var min1, min2 :  minxy);
begin
   if min1.d < min2.d then
      min2 := min1;
   end;
end;

         ⋮

...  minloc<< (A, Index1, Index2) ...;
```

Figure 3: User-de£ned `minloc` reduction in ZPL

where, under any reasonable condition, work is done. This offers the ZPL programmer enough ¤exibility to write an ef£cient code.

4.4 Associativity, commutativity, and aggregation

User-de£ned reductions in ZPL must be associative to guarantee determinism and a correct answer. The programmer is responsible for verifying that this is the case. This is consistent with other languages that support user-de£ned reductions such as C** and SAC. In C** and SAC, user-de£ned reductions must also be commutative. For ZPL, we have weakened this condition and assume instead that a user-de£ned reduction is not commutative. This decision is pending results from a performance study. Whereas associativity is necessary to use the parallel-pre£x method and exploit parallelism, commutativity is not. Commutativity is advantageous only to certain parallel computers that can take advantage of values arriving in different orders. Non-commutative reductions are common. For example, given a one-dimensional array of ones and zeroes, the length of the longest sequence of ones can be determined in parallel using an associative, non-commutative reduction.

Aggregation is an important method for limiting the number of messages sent in message passing systems in the presence of many similar reductions. It has proven vital to achieving high performance [14]. If given a parallel array where each element corresponds to a list of k elements, we want to £nd the smallest elements that reside in each position of the list, we would write k reductions in a loop. These k reductions would be aggregated by the ZPL compiler. Aggregation occurs in the ZPL case for user-de£ned reductions just as for built-in reductions.

In the code in Appendix A, a signi£cant difference can be seen between the global functions in the C+MPI and ZPL implementations. In the C+MPI implementation, the global function takes an array of reduction elements rather than a single one. In ZPL, the global function is passed just a single element. However, the ZPL compiler transforms the global function into one that takes an array of elements and aggregation is done automatically when applicable.

5 Evaluation

To determine the effect on performance of user-de£ned reductions, we ran three versions of the NAS MG benchmark [1, 2] on a 272 processor T3E-900. Each processor runs at 450MHz and there are 256 MB of RAM per processor. The three versions of the NAS MG benchmark are the original NAS implementation in F77 and MPI, a ZPL implementation using only built-in reductions, and a ZPL implementation using the user-de£ned reduction mechanism described in this paper.

Our focus is on the initialization of the array in the NAS MG benchmark which works as follows. First, the array is £lled with random numbers. Second, the ten largest and ten smallest values are identi£ed. Third, these twenty values are replaced with the values $+1$ and -1 respectively and all other values in the array are set to zero. It is assumed that the ten largest and ten smallest values are unique. In our timings, we focus only on the second step of this process. Figure 4 contains the results of our experiment on the three large classes. Note that classes A and B are identical with regards to the initialization of the array.

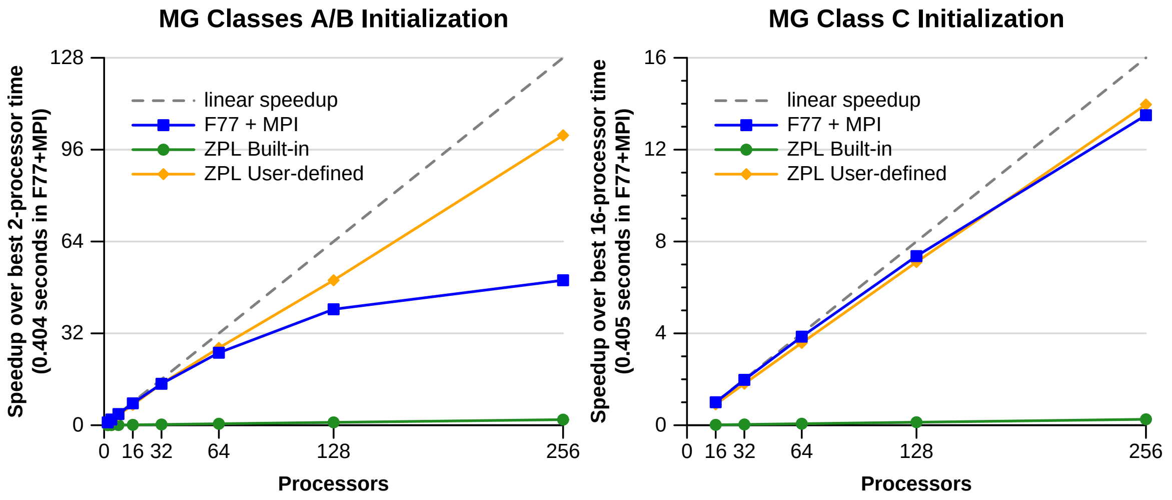

Figure 4: Parallel speedup of NAS MG's initialization

The graphs in Figure 4 show us that the addition of user-de£ned reductions in the ZPL language is critical to performance. Although the ZPL implementation using only built-in reductions scales almost linearly with respect to itself, the overhead is too high. In this implementation, we compute 20 reductions. Each reduction returns a minimum or maximum value in the array. Between these reductions, each processor scans through its portion of the array to £nd the reduced value. If it is found, the location is determined and this information is broadcast to all processors. The amount of computation is overwhelming; the array is traversed a total of 40 times. In the ZPL implementation with user-de£ned reductions as well as the F77+MPI implementation, the array is traversed only once.

Another point to note from the graphs is that the ZPL implementation with user-de£ned reductions is only slightly slower than the F77+MPI implementation on a small number of processors (within 10%) and that on a large number, the ZPL implementation continues to scale whereas the F77+MPI implementation does not. This continued scaling is a result of implementation differences; it does not re¤ect a limitation of expressibility in the F77+MPI implementation.

The F77+MPI implementation avoids user-de£ned reductions by taking advantage of the local view of the computation. In a single traversal over the array, each processor £nds the ten smallest and ten largest values that reside on its portion of the array. After this, twenty reductions are used to £nd which values are the largest and smallest in the entire array. No location information need be broadcast. If a processor's locally largest values are globally largest, then this processor knows that it will replace these values with +1. This is in contrast to what can be done in ZPL.

In both ZPL implementations, the location information of the largest and smallest values in the entire array must be broadcast to each processor to maintain the global view of the computation. Our ZPL implementation with user-de£ned reductions scales better than the F77+MPI reduction only because we use a single, large reduction to £nd the twenty globally largest and smallest values in the array. On smaller processors, the ZPL implementations suffer from the extra overhead involved in communicating and using the location information on a global scale.

6 Related Work

The idea of a construct for user-de£ned reductions is not new, though it remains surprisingly absent from many high-level languages. When they are supported, it is often not as ef£cient as possible. They are supported in SAC [16], but in a limited form. Only one function is speci£able for both the global and local parts of the reduction. This makes reductions like the `minten` reduction dif£cult to write in an ef£cient manner for reasons discussed in Section 4.2.

Viswanathan and Larus [20] developed a powerful mechanism for user-de£ned reductions in the context of the C** language that closely resembles the construct described in this paper. However, they provide no mechanism for passing extra parameters to the local function and it is unclear as to how the initialization phase is done, whether with another overloaded function or not. In addition, due to language differences, their mechanism for user-de£ned reductions can lead to data races. The higher-level, global view of the computation in ZPL eliminates this worry.

7 Conclusion

The optimized handling of reductions on parallel supercomputers or clusters of workstations is critical to high performance because reductions are common in scienti£c codes and a potential source of bottlenecks. Consequently, researchers have worked diligently on techniques for compilers and programmers to use so that reductions execute ef£ciently. Great strides have been made in the domain of automatic parallelization, but this remains a hit-and-miss approach to high performance. Semi-automatic techniques relying on directives have improved the hit rate, but performance still often suffers. And so the use of a language like Fortran 77 coupled with a message passing library like MPI remains the popular standard. Consistently high performance is crucial to most scienti£c programmers who are willing to expend the considerable effort necessary to program with Fortran 77 and MPI. Message passing libraries are dif£cult to use; they force the programmer to write programs on a per-processor basis, to tediously engineer all interprocessor communication, and to lose track of the problem as a whole.

High-level parallel languages are a promising alternative to the popular standard, but even for the well-studied idiom of reductions, performance suffers. This is because what can be directly done by a programmer with Fortran 77 and MPI can often not be done in any given high-level language. There is either no mechanism for user-de£ned reductions or there is one that forces the programmer to sacri£ce some amount of ef£ciency. In this paper, we have presented a new language construct for arbitrary reductions that lets a programmer achieve a level of high performance equal to that achievable with Fortran and MPI. We evaluated our approach in the context of the NAS MG benchmark and showed that performance closely resembles that achieved with the low-level Fortran plus MPI approach. This construct is vital to high performance and makes high-level languages a more viable choice for scientists.

Acknowledgments

The £rst author is supported by a DOE High-Performance Computer Science Fellowship and completed a portion of this work while at Los Alamos National Laboratory. We would like to thank Sung-Eun Choi and our anonymous reviewers for their many insightful comments on earlier drafts of this paper. This work was supported in part by a grant of HPC resources from the Arctic Region Supercomputing Center.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. NAS parallel benchmarks. Technical report, NASA Ames Research Center (RNR-94-007), March 1994.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. NAS parallel benchmarks 2.0. Technical report, NASA Ames Research Center (NAS-95-020), December 1995.

[3] G. E. Blelloch. NESL: A nested data-parallel language (Version 3.1). Technical report, Carnegie Mellon (CMU-CS-95-170), September 1995.

[4] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoe¤inger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1994.

[5] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

[6] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the ACM Conference on Supercomputing*, 2000.

[7] B. L. Chamberlain, E. C. Lewis, C. Lin, and L. Snyder. Regions: An abstraction for expressing array computation. In *Proceedings of the ACM International Conference on Array Programming Languages*, 1999.

[8] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1994.

[9] High Performance Fortran Forum. *High Performance Fortran Language Speci£cation, Version 2.0.* 1997.

[10] O. Ibarra, M. C. Rinard, and P. C. Diniz. On the complexity of commutativity analysis. In *Proceedings of the International Computing and Combinatorics Conference*, 1996.

[11] R. E. Ladner and M. J. Fischer. Parallel pre£x computation. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1977.

[12] J. R. Larus, B. Richards, and G. Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical report, University of Wisconsin-Madison (1126), November 1992.

[13] J. R. Larus, B. Richards, and G. Viswanathan. Parallel programming in C**: A large-grain data-parallel programming language. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, Cambridge, MA, USA, 1996. MIT Press.

[14] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the International Parallel Processing Symposium*, 1998.

[15] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1996.

[16] S.-B. Scholz. On de£ning application-speci£c high-level array operations by means of shape-invariant programming facilities. In *Proceedings of the ACM International Conference on Array Programming Languages*, 1998.

[17] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.

[18] L. Snyder. *Programming Guide to ZPL.* MIT Press, Cambridge, MA, USA, 1999.

[19] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the ACM International Conference on Supercomputing*, 1996.

[20] G. Viswanathan and J. R. Larus. User-de£ned reductions for ef£cient communication in data-parallel languages. Technical report, University of Wisconsin-Madison (1293), January 1996.

A User-de£ned Reduction in C+MPI

The following snippets of code contain an implementation of the `minloc` reduction in C+MPI. Recall that in the `minloc` reduction, we £nd not only the minimum element in an array, but also its location. It should be noted that this particular user-de£ned reduction might be more easily implemented in C+MPI using MPI's built-in MINLOC intrinsic and an appropriate mapping between the location coordinates used in this example and a single integer. However, we do not use this mechanism so as to demonstrate MPI's mechanism for dealing with user-de£ned reductions. This mechanism must be used for reductions such as the `minten` reduction discussed in this paper.

```
typedef struct {
   double d;
   int x, y;
} minxy;

void minloc(minxy* in, minxy* inout, int* len,
            MPI_Datatype *dptr) {
   int i;
   for (i = 0; i < *len; ++i) {
      if (inout->d > in->d) {
         *inout = *in;
      }
      in++; inout++;
   }
}

          ⋮

int i, base;
minxy ml;
MPI_Op myOp;
MPI_Datatype type[3] = MPI_INT, MPI_INT, MPI_INT;
MPI_Aint disp[3];
int blocklen[3] = 1, 1, 1;
MPI_Datatype mpiml;
MPI_Address(ml, disp);
MPI_Address(ml.x, disp+1);
MPI_Address(ml.y, disp+2);
base = disp[0];
for (i = 0; i < 3; ++i) disp[i] -= base;
MPI_Type_struct(2, block, disp, type, &mpiml);
MPI_Type_commit(&mpiml);
MPI_Op_create(minloc, True, &myOp);

          ⋮

int i, j;
minxy minl, gminl;
for (i = 0; i < local_high_ni; i++) {
   for(j = 0; j < local_high_nj; j++) {
      if (a[i][j] < minl.d) {
         minl.d = a[i][j];
         minl.x = i;
         minl.y = j;
      }
   }
}
MPI_Allreduce(&minl, &gminl, 1, mpiml, myop);
```