Global-View Abstractions for User-Defined Reductions and Scans *

Steven J. Deitz David Callahan
Bradford L. Chamberlain
Cray Inc.
{deitz,bradc}@cray.com,d.callahan@microsoft.com

Lawrence Snyder
University of Washington
snyder@cs.washington.edu

Abstract

Since APL, reductions and scans have been recognized as powerful programming concepts. Abstracting an accumulation loop (reduction) and an update loop (scan), the concepts have efficient parallel implementations based on the parallel prefix algorithm. They are often included in high-level languages with a built-in set of operators such as sum, product, min, etc. MPI provides library routines for reductions that account for nearly nine percent of all MPI calls in the NAS Parallel Benchmarks (NPB) version 3.2. Some researchers have even advocated reductions and scans as the principal tool for parallel algorithm design.

Also since APL, the idea of applying the reduction control structure to a user-defined operator has been proposed, and several implementations (some parallel) have been reported. This paper presents the first *global-view* formulation of user-defined scans and an improved global-view formulation of user-defined reductions, demonstrating them in the context of the Chapel programming language. Further, these formulations are extended to a message passing context (MPI), thus transferring global-view abstractions to *local-view* languages and perhaps signaling a way to enhance local-view languages incrementally. Finally, examples are presented showing global-view user-defined reductions "cleaning up" and/or "speeding up" portions of two NAS benchmarks, IS and MG. In consequence, these generalized reduction and scan abstractions make the full power of the parallel prefix technique available to both global- and local-view parallel programming.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—concurrent, distributed and parallel languages

General Terms Languages

Keywords parallel programming, reductions, scans, parallel prefix, MPI, Chapel

* This work was funded in part by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003 and was supported in part by a grant of HPC resources from the Arctic Region Supercomputing Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'06 March 29–31, 2006, New York, New York, USA.
Copyright © 2006 ACM 1-59593-189-9/06/0003. . . $5.00.

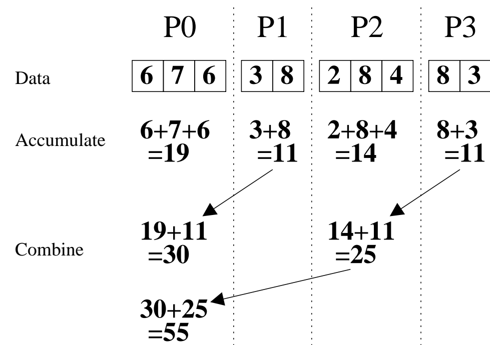

Figure 1. An illustration of a parallel reduction divided into an accumulate phase and a combine phase. In the accumulate phase, the processes independently compute partial sums. In the combine phase, the processes communicate (indicated by the arrows) to add together the local sums and compute the total sum.

1. Introduction

Reductions and scans are useful primitives for parallel computing because they map well to both application and architecture. In the NAS Parallel Benchmarks (NPB) version 3.2 [1], nearly 9% of the MPI calls are reductions. Reductions have been supported with special-purpose hardware on some parallel systems, *e.g.*, CM-5, and scans are efficiently implemented by the parallel-prefix algorithm [11]. So ubiquitous are the operations abstracted by reductions and scans that Blelloch has advocated them as the principal abstractions for parallel computation [3].

> **Definition.** The **reduction operation** takes a binary operator $\oplus$ and an ordered set of values $[a_1, a_2, \ldots, a_n]$ and returns the value
>
> $$a_1 \oplus a_2 \oplus \cdots \oplus a_n.$$

For example, if $\oplus$ is addition then the reduction of the ordered set $[6, 7, 6, 3, 8, 2, 8, 4, 8, 3]$ is 55.

If the $\oplus$ operator is associative then an efficient parallel implementation exists for the reduction. Figure 1 illustrates a parallel execution of the sum reduction in two phases. In the accumulate phase, the local data is summed to form a local sum. Then in the combine phase, these partial sums are added together to form the total sum. In the figure, the arrows identify communication that generalizes to a log tree.

If the $\oplus$ operator is commutative, there is potential for a more efficient implementation than if the $\oplus$ operator is non-commutative. If the branching factor on the log tree is greater

than two (common for many parallel machines), then reductions of commutative operators can immediately combine whichever partial results are available whereas reductions on non-commutative operators must stick to a predefined order. Additionally, a commutative operator allows for potentially taking better advantage of the network by routing the values based on the physical location of the processors without concern for the order of the set.

Operators that are non-associative must be performed in the order specified by the semantics of the language, thus limiting the potential for parallelism. Although programmers tend to think of applying reduce and scan to operators that are associative, applying them to non-associative operators still has the advantage of giving a more abstract statement of the computation.

Definition. The **scan operation** takes a binary operator $\oplus$ and an ordered set of n values $[a_1, a_2, \ldots, a_n]$ and returns the ordered set of values

$$[a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_n].$$

The **exclusive scan operation** returns instead the ordered set of values

$$[i, a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}]$$

where i is the identity of $\oplus$.

For example, if $\oplus$ is addition then the scan of the ordered set $[6, 7, 6, 3, 8, 2, 8, 4, 8, 3]$ is

$$[6, 13, 19, 22, 30, 32, 40, 44, 52, 55]$$

and the exclusive scan is

$$[0, 6, 13, 19, 22, 30, 32, 40, 44, 52].$$

The standard definition of scan can be called the *inclusive* scan because it includes the initial element. Exclusive scan is desirable because it enables the elegant recursive definitions of multi-dimensional scans. Notice also that the inclusive scan can be defined in terms of the exclusive scan. The elements in the set produced by the inclusive scan can be computed by applying the $\oplus$ operator to the elements in the original set and the elements in the set produced by the exclusive scan. The content of this paper is unaffected by which definition is chosen as the standard.

The subject of this paper is global-view abstractions for user-defined reductions and scans. To give some intuition behind the difference between local-view and global-view abstractions, reconsider Figure 1. At the local-view level, the user must think about the per-processor code. The abstraction provided thus only applies to the combine phase of the reduction. At the global-view level, the reduction applies to both the combine and accumulate phases.

This paper's contributions are as follows:

- It extends previously proposed abstractions for user-defined reductions [9] and adds support for user-defined scans. It demonstrates how these abstractions can be made more flexible and robust in a modern object-oriented language such as Chapel.
- It adapts this global-view abstraction to MPI, raising the level of abstraction for user-defined reductions and scans in MPI. This suggests the potential for incremental improvements to lower level approaches to parallel programming by appropriating ideas from higher level global-view languages.
- It illustrates the abstraction with many examples and identifies two places in the NAS kernel benchmarks where user-defined reductions can improve readability and sometimes performance.

The combined message of these results is the extension of clean and efficient mechanisms for reductions and scans to both global and local parallel programming practice, resulting in both greater clarity and (often) better performance.

This paper is organized as follows. The next section describes a local-view abstraction for user-defined reductions and scans. It shows how MPI implements it. Section 3 describes the global-view abstraction for user-defined reductions and scans. It shows how Chapel implements it. Section 4 introduces RSMPI, an extension to MPI, which implements the global-view abstraction for use in an MPI program. This section also evaluates the abstraction in the context of RSMPI and rewrites of the NAS benchmarks. It shows shorter, better abstracted code, and comparable or improved performance. Section 5 discusses related work, and Section 6 summarizes the results of this paper.

2. Local-View Abstraction

Local-view parallel programming is the most common form of parallel programming in use today. In this model, each processor executes a separate copy of the program independently, communicating with one another via some mechanism. The end result to this model is that the programmer must manage explicitly what each processor is doing at all times and how the data is distributed and communicated.

A local-view reduction or scan is relatively straightforward to implement. The abstraction assumes each processor has one data value for each resulting value. In Figure 1, this means that the accumulate phase of the reduction has already been computed on each processor, and now the processors are ready to combine the partial sums. Note that each processor may have accumulated more than one data value, but then the end result of the reduction is also more than one data value. We discuss this technique of aggregating reductions later.

In order to implement a user-defined operator for the local-view abstraction, the user must define two functions: an identity function and a combine function. The identity function computes the identity of the operator, and the combine functions computes the reduction of two values. For a reduction of multiple values, the combine function can be used in the implementation of the parallel log tree as illustrated by Figure 1.

The local-view abstractions can be supported by four routines. Two reduction routines, LOCAL_ALLREDUCE and LOCAL_REDUCE, compute a reduction and, respectively, leave the result on all of the processors or a single processor. These routines take two arguments: the combine function and a single value on each processor that is to be reduced. Two scan routines, LOCAL_XSCAN and LOCAL_SCAN, compute exclusive or inclusive scans respectively. These routines take three arguments, the extra argument being the identity function, which is necessary for the exclusive scan.

As an example, Listing 1 shows the two functions one would write in C to implement the **min**k reduction. In this reduction, each processor starts with a vector of k elements in sorted order from high to low. The reduction combines these values so that the result contains the k minimum values in all of the vectors.

Note that the inclusive scan is less important than the exclusive scan because it can be computed (without communication) given the exclusive scan. The converse is not true. Given the inclusive scan, it is impossible to compute the exclusive scan without communication if the combine function cannot be inverted. For example, a function that computes the minimum of two values cannot be inverted. In this case, the exclusive scan can only be computed from the inclusive scan by shifting the values across the processors.

2.1 Aggregation

Aggregation is an important extension to the local-view reduction. It allows the programmer to compute multiple reductions simulta-

Listing 1. The min$k$ operator in C.

```
void ident(int v[]) {
  int i;
  for (i = 0; i < k; i++)
    v[i] = INT_MAX;
}

void combine(int v1[], int v2[]) {
  int i, j, tmp;
  for (i = 0; i < k; i++)
    if (v1[i] < v2[0]) {
      v2[0] = v1[i];
      for (j = 1; j < k; j++)
        if (v2[j-1] < v2[j]) {
          tmp = v2[j];
          v2[j] = v2[j-1];
          v2[j-1] = tmp;
        }
    }
}
```

neously, thus saving the overhead of many smaller messages. The local-view routines discussed above can be augmented with an extra argument for the number of values each processor is reducing. In addition, arrays of the values must be passed to the routines rather than just the values.

For example, the min reduction can be aggregated to compute the element-wise minimums of the values in arrays of integers. Note that this aggregation is different from the user-defined min$k$ reduction in that the element-wise minimums are computed instead of the overall minimums. Indeed, the min$k$ reduction can itself be aggregated to compute the element-wise k minimums of the values in arrays of vectors.

2.2 MPI Constructs

MPI provides twelve built-in operations for reductions and scans: maximum, minimum, sum, product, logical and, bit-wise and, logical or, bit-wise or, logical xor, bit-wise xor, maximum value and location, and minimum value and location. It allows the user to create user-defined operations in the form of a combine function that is extended for aggregation.

MPI provides routines that correspond exactly to the local-view routines described earlier. However, MPI does not require an identity function. Instead, the first element in the result of an exclusive scan is undefined.

3. Global-View Abstraction

The global-view abstraction computes both the accumulate and combine phases of the reduction in Figure 1. It takes a more global view of the computation by pushing the per-processor code into the abstraction. For example, the min$k$ reduction used as an example for the local-view abstraction would look different when implemented with the global-view abstraction. Given an array of n integers distributed over p processors where n is much larger than p, the user-defined min$k$ reduction computes the k minimum integers. In the local-view abstraction, the programmer computes sorted vectors of the k minimums before calling into the reduction.

The global-view abstraction allows for the type of the input values to be different than the type of the output value. In the case of the min$k$ operator, the input type is an integer and the output type is a vector of k integers. The abstraction works by defining separate functions for the accumulate and combine phases. In addition to allowing the input type and the output type to be different, the type

Listing 2. Algorithm for the global-view reduction.

```
forall processors q in 0..p-1
  s_q ← f_ident()
  if n > 0
    s_q ← f_pre_accum(s_q, in_q(0), ...)
  for i in 0..n-1
    s_q ← f_accum(s_q, in_q(i), ...)
  if n > 0
    s_q ← f_post_accum(s_q, in_q(n-1), ...)
LOCAL_REDUCE(f_combine, s_q)
forall processors q in 0..p-1
  out_q ← f_red_gen(s_q)
```

of the state—the value that is accumulated into and then passed between the processors—may also be different.

If the input type, output type, and state type are the same, then the global-view abstraction reduces to the local-view abstraction. The identity function, f_{ident}, and combine function, $f_{combine}$, need to be specified by the programmer. The combine function is then used to accumulate the values into a local result. The result of the combine function is then returned.

If the input type is different, then an accumulate function, f_{accum}, must also be specified. These functions have the following type signatures if the input type is in and the output type is out:

$$f_{ident} : () \rightarrow out$$
$$f_{accum} : (in \times out) \rightarrow out$$
$$f_{combine} : (out \times out) \rightarrow out$$

If the output type is different than the state type, then a generate function must also be specified. For a reduction, the generate function returns the single value from a final state value. For a scan, it returns a value from each of the states. These functions are called f_{red_gen} and f_{scan_gen} respectively. They have the following type signatures where the state type is $state$:

$$f_{ident} : () \rightarrow state$$
$$f_{accum} : (in \times state) \rightarrow state$$
$$f_{combine} : (state \times state) \rightarrow state$$
$$f_{red_gen} : (state) \rightarrow out$$
$$f_{scan_gen} : (in \times state) \rightarrow out$$

Note that the scan generator can produce a different value based on the input value at each position.

In addition to these functions, it is useful to allow a function to act on the state based on the first value on the processor before accumulating and another function to act on the state based on the last value on the processor after accumulating. These functions are called f_{pre_accum} and f_{post_accum} and have the same type signature as f_{accum}. Listing 7 illustrates a situation where these functions are useful.

Listings 2 and 3 outline the algorithm for computing the global-view reduction and scan in terms of the local-view abstractions. Note that the global-view scan listing computes an exclusive scan. By interchanging lines 12 and 13, this algorithm is made to compute an inclusive scan.

The accumulate function often has a substantially faster implementation than the combine function, and it should be optimized at the combine function's expense. The functions defined in this section take advantage of this property. Alternative functions that translate the input values into state values rather than accumulate the input values into state values would result in worse performance.

3.1 Examples in Chapel

Chapel is a new parallel programming language being developed by Cray Inc. in conjunction with Caltech/JPL as part of DARPA's

Listing 3. Algorithm for the global-view exclusive scan.

```
1  forall processors q in 0..p-2
2    s_q ← f_ident()
3    if n > 0
4      s_q ← f_pre_accum(s_q, in_q(0), ...)
5    for i in 0..n-1
6      s_q ← f_accum(s_q, in_q(i), ...)
7    if n > 0
8      s_q ← f_post_accum(s_q, in_q(n-1), ...)
9  LOCAL_XSCAN(f_ident, f_combine, s_q)
10 forall processors q in 0..p-1
11   for i in 0..n-1
12     out_q(i) ← f_scan_gen(s_q, in_q(i), ...)
13     s_q ← f_accum(s_q, in_q(i), ...)
```

Listing 4. An implementation of the min$k$ operator in Chapel.

```
1  class mink {
2    type in_t;
3    const k : integer;
4    var v: [1..k] in_t = in_t.max;
5    function accum(x: in_t)
6      if x < v[1] {
7        v[1] = x;
8        for i in 2..k
9          if v[i-1] < v[i] {
10           var tmp = v[i];
11           v[i] = v[i-1];
12           v[i-1] = tmp;
13         }
14     }
15   function combine(s: mink(in_t))
16     for i in 1..k
17       accum(s.v[i]);
18   function gen()
19     return v;
20 }
```

HPCS (**H**igh-**P**roductivity **C**omputing **S**ystems) program. Chapel allows the programmer a more global view of the computation and is well-suited to the global-view abstraction presented in this paper. As an object-oriented language that supports generic programming and type inference, Chapel improves the abstraction in some non-trivial ways.

3.1.1 Example 1: min*k*

Listing 4 shows an implementation of the min$k$ operator in Chapel. The operator is a class that implements an interface allowing it to be used as the operator for a reduction or scan. To make the operator more flexible, it is parameterized by the type of the values being reduced or scanned; Line 2 names this type `in_t`. The state is stored in the fields of the class. Here the state is the array `v` of size `k`.

The default constructor computes the identity. In Chapel, a default constructor is created for all classes if one is not specified. The fields are then initialized. So the default constructor for the `mink` class initializes the elements of `v` to the maximum value of the input type.

Every class that defines an operator for use in a reduction or scan must define at least the three functions `accum`, `combine`, and `gen`. This is necessary in Chapel because the state type is necessarily different from the input and output types since it is the class itself. However, if the input and state types are conceptually the same, the `combine` function can be implemented to simply call the `accum` function.

Listing 5. An implementation of the min$i$ operator in Chapel.

```
1  class mini {
2    type elt_t;
3    var val: elt_t = elt_t.max;
4    var loc: integer;
5    function accum(x: (elt_t,integer))
6      if x(1) < val then
7        (val, loc) = x;
8    function combine(s: mini(elt_t))
9      call accum((s.val, s.loc));
10   function gen()
11     return (val, loc);
12 }
```

In the implementation, the state used during the reduction is of the type of the class. The state is then the implicit `this` argument to the methods. Notice that the second state is explicitly passed into the combine function as the function's only argument. Its type is specified to be the class instantiated by the same type, namely, `in_t`.

The result of the `mink` reduction is the array of `k` minimum values. To call this reduction in Chapel over an array of integers `A`, the programmer declares the array of k integers that is returned from the reduction and calls the reduction directly on `A` as in

```
var minimums: [1..10] integer;
minimums = mink(integer,10) reduce A;
```

The compiler transforms this code based on the `mink` class. It creates as many instances of that class as are needed based on how many processes are used. The details of the reduction are determined by the functions that are provided by the class. In this case, there are no `pre_accum` or `post_accum` functions, so lines 3 and 4 from Listing 2 are omitted. Also note that the `gen` function is used instead of `red_gen` since there is no separate `red_gen` function. In many cases, reductions and scans can share the same generate functions.

3.1.2 Example 2: min*i*

The min$i$ operator finds the minimum value and its location. Listing 5 shows an implementation of the min$i$ operator in Chapel. Chapel's support for first-class tuples allows multiple values to be passed into the accumulate function. Given an array of integers `A`, the minimum and its location can be found via the `mini` reduction by writing

```
var (val, loc) =
   mini(integer) reduce
     [i in 1..n] (A(i), i);
```

In this code, the result of the reduction is stored in two variables, `val` and `loc`. The source of this reduction is an array expression created by `[i in 1..n]`. Conceptually, this expression creates an array of `n` tuples though the array may not be allocated depending on the implementation.

3.1.3 Example 3: counts

Given a list of particles with locations in one of eight octants, a reduction could determine how many particles are in each location. A scan could determine a ranking of the particles within each octant. For example, if ten particles are located in octants 1 through 8 based on the ordered set $[6, 7, 6, 3, 8, 2, 8, 4, 8, 3]$, then the reduction would return the ordered set of eight counts

$$[0, 1, 2, 1, 0, 2, 1, 3]$$

Listing 6. An implementation of the counts operator in Chapel.

```
1  class counts {
2    var v: [1..k] integer;
3    function accum(x: integer) {
4      v[x] += 1;
5    }
6    function combine(s: counts) {
7      v += s.v;
8    }
9    function red_gen()
10      return v;
11   function scan_gen(x: integer)
12      return v[x];
13 }
```

Listing 7. An implementation of the sorted operator in Chapel.

```
1  class sorted {
2    type in_t;
3    param commutative = false;
4    var status: boole = true;
5    var first: in_t = in_t.max;
6    var last: in_t = in_t.min;
7    function pre_accum(x: in_t) {
8      first = x;
9    }
10   function accum(x: in_t) {
11     if last > x then
12       status = false;
13     last = x;
14   }
15   function combine(s: sorted(in_t)) {
16     status = status and s.status
17                  and last <= s.first;
18     last = s.last;
19   }
20   function gen()
21      return status;
22 }
```

and the scan would return the rankings

$$[1, 1, 2, 1, 1, 1, 2, 1, 3, 2].$$

.

This operator uses a different generate function depending on whether it is being used in a reduction or scan. Listing 6 shows an implementation of the counts operator in Chapel.

3.1.4 Example 4: sorted

Given an ordered set of values, the sorted reduction determines whether the values are in sorted order. This non-commutative reduction keeps track of the first and last elements in the accumulate function. The combine function then ensures that each of the parts that are accumulated are sorted and that the boundary elements are in sorted order as well.

Listing 7 shows an implementation of the sorted operator in Chapel. This class specifies that the operator is non-commutative by specifying the "`param`" in line 3. This compile-time constant allows different reduction algorithms to be selected. If it is undefined, it is assumed to be true by the compiler.

4. RSMPI: An Extension to MPI

This section proposes adding support for global-view scans and reductions to MPI. It does this by proposing an extension to MPI,

Listing 8. An implementation of the sorted operator in C+RSMPI.

```
1  rsmpi operator sorted {
2    non-commutative
3    state {
4      int first, last;
5      int status;
6    }
7    void ident(state s) {
8      s->first = INT_MAX;
9      s->last = INT_MIN;
10     s->status = 1;
11   }
12   void pre_accum(state s, int i) {
13     s->first = i;
14   }
15   void accum(state s, int i) {
16     if (s->last > i)
17       s->status = 0;
18     s->last = i;
19   }
20   void combine(state s1, state s2) {
21     s1->status &= s2->status &&
22       (s1->last <= s2->first);
23     s1->last = s2->last;
24   }
25   int generate(state s) {
26     return s->status;
27   }
28 }
```

called RSMPI (Reduce and Scan MPI), that requires a simple preprocessor to convert RSMPI code to MPI code. This section explains how RSMPI works, discusses how its usage cleans up MPI code in the context of the NAS benchmarks, and shows some performance results that demonstrate its advantages.

Since RSMPI is transformed to MPI, it is always possible to write MPI that is as fast as RSMPI. The global-view abstraction in RSMPI, however, makes it significantly easier to use. It makes it possible to build up a library of operators that compute an entire reduction or scan, not just the combine portion.

RSMPI faithfully implements the global-view abstraction so an example is sufficient to explain it. Listing 8 shows an implementation of the sorted operator in C+RSMPI. Superficial changes made by a preprocessor translate this code into a set of functions that can then be used at the call-site.

For example, to use the sorted operator to determine if an array of integers is in sorted order, the programmer first defines an iterator to describe the values passed to the accumulate function and then calls an RSMPI routine to reduce or scan.

```
rsmpi iterator values {
  for (i=0; i<n; i++)
    yield i;
}

RSMPI_Reduceall(status, sorted,
                values, A[values]);
```

The call to RSMPI_Reduceall takes the result, the RSMPI operator, the RSMPI iterator, and the input expression. The iterator identifies the values that each processor needs to accumulate. The accumulate function is applied to the input expression within this iterator and then inlined into the code.

The RSMPI routine has similar structure to the MPI routine, though we allow the common case of using the MPI_COMM_WORLD communication group as a default if another is omitted.

To evaluate the use of RSMPI, we studied the NAS benchmarks and found two places where user-defined reductions could significantly simplify the code that the programmer has to write in the main body of the program. These areas of the code were not in the parts of the benchmark that are timed, but were instead in the initialization and verification phases. They represent idioms that show up in practice.

4.1 NAS IS: An RSMPI Case Study

As the last part of the computation, the NAS IS benchmark verifies that the large array of integers is sorted. Rather than using a user-defined reduction like the sorted reduction described in Section 3.1.4, this is written in MPI in the following way. First, the boundary elements are communicated to neighboring processors so that it can be determined that the boundary elements are in sorted order. Then, locally on each processor, all the other elements are checked to make sure they are in sorted order. Finally a sum reduction is used to determine that all of the processors have sorted values.

Using MPI's user-defined reduction mechanism, this can be marginally improved. In the program flow, however, the programmer must still specify the local portion of the reduction. This is awkward compared to using the global-view abstraction of RSMPI. With this abstraction, a single line can apply the sorted reduction to the conceptual entire array of integers.

Figure 2 shows timings of the C+MPI version of NAS IS compared to a C+RSMPI version. The C+RSMPI code was generated by an experimental prototype of an RSMPI preprocessor written in Perl.

The RSMPI version performs better based on a scalar improvement. Since values in the array are stored in a scalar when being compared to their neighbors, only one memory reference is made per value in the array. Optimizing the provided NAS C+MPI code to make one memory reference per value in the array closed the performance gap entirely.

The difference between the parallel RSMPI and MPI version is non-trivial, though our timings revealed no difference. In the RSMPI version, the reduction requires larger messages than in the MPI version and it is non-commutative. The MPI version, on the other hand, requires an initial message to be passed between neighboring processors.

In an experiment to see whether any gains would be made if the user-defined reduction were commutative, we flagged the reduction as commutative. This resulted in no speedup, though the program did fail to verify that the array was sorted (as expected).

4.2 NAS MG: An RSMPI Case Study

In the initialization of the NAS MG benchmark, an array is filled with random numbers. The ten largest numbers and their locations in the array along with the ten smallest numbers and their locations in the array are then identified. These positions are then filled with positive ones and negative ones respectively, and the rest of the array is filled with zeros.

In the NAS F+MPI version, this portion of the computation, the ZRAN3 routine, is implemented with forty reductions. In the F+RSMPI version, a single user-defined reduction, similar to the min$k$ and min$i$ reductions described previously, can be used instead. Figure 3 shows timings of the F+MPI version of NAS MG compared to a hand-coded F+RSMPI version. The overhead of not using the single user-defined reduction is seen more sharply in smaller problem classes since the reduction accounts for more of the time. In larger class sizes, more time is taken to traverse and compute on the array, so the efficiency is more comparable.

5. Related Work

The importance of user-defined scans and reductions is well-established. Parallelizing these constructs from sequential code has garnered much attention. Fisher and Ghuloum [10] extended simple techniques for recognizing standard reductions and scans to handle more complicated operators by analyzing conditional branches in loops.

Despite the importance of scans and reductions to parallel computing, however, language support for user-defined variants are lacking from most parallel programming approaches. MPI is a notable exception that has already been discussed. Other communication libraries vary in their support of user-defined scans and reductions. PVM [14] supports user-defined reductions similar to MPI, but has no support for user-defined scans. ARMCI [12] and SHMEM [2] provide a small set of built-in reductions, but no user-defined reductions and no scans.

Support for collective operations is lacking in Co-Array Fortran [13], Titanium [16], and UPC [5]. Arguments for adding collective communication have been made in the literature [6].

The NESL language provides support for many built-in scans and reductions, but provides no support for user-defined scans or reductions [4]. This language is novel in its heavy reliance on scans, demonstrating how effective this primitive can be.

The C** language provides good support for global-view user-defined reductions [15] similar to those described in this paper. However, they do not support identity or generate functions, nor do they support user-defined scans.

MapReduce [7] is a recent programming model used to apply an operation and a reduction to a large data set. The system is capable of taking advantage of associativity by letting the user specify a "combine" function alongside a "reduce" function. These parallel our "accumulate" and "combine" functions. The MapReduce model follows a different implementation path than the mechanisms discussed here because of its design point (querying a large database as opposed to scientific computing).

Previous work on support for user-defined reductions in ZPL was based on overloading procedures of the same name in order to specify the identity, accumulate, and combine functions [9]. (There was neither a generate function nor support for user-defined scans.) There were three main problems with the approach of overloading procedures. First, the error messages that arose when the procedures were not defined exactly as they should be were often misleading or uninformative. Second, the user-defined reductions could only be used in simple assignment statements so that the correct result type could be determined. Third, if the state type was the same as the initial type, there was no way of supplying a combine function that was different than the accumulate function. Although this is not common, it seems possible given a fairly complex reduction and initial type. Later work sketched out support in ZPL for a generate function and user-defined scans [8]. This work was not in an object-oriented framework and was substantially bulkier as a result.

6. Conclusion

Reduce and scan are clean, high-level programming abstractions that admit efficient parallel implementations. Although it has been common for languages to provide built-in reduce and scan for a small set of associative and commutative operators, programmers have long recognized that the idea is more general.

This paper has formulated the principle of parallel prefix computation in a way that allows users to apply it practically in service of their own scan and (degenerately) reduce. The results apply to both global- and local-view parallel programming situations. Specifically, we present the first user-defined scan formulation for

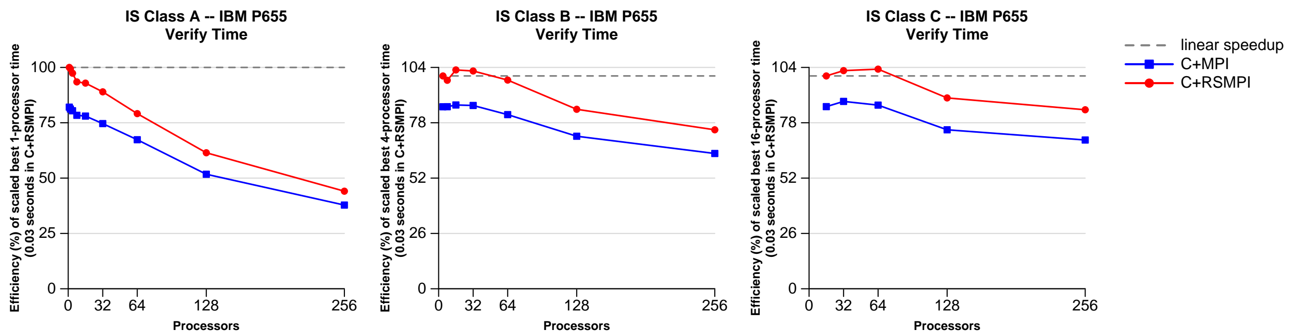

Figure 2. Efficiency graphs showing the speedup of the verification phase of classes A, B, and C of the NAS IS benchmark on an IBM P655 system with 92 nodes, each containing eight 1.5 GHz power4 processors and 16 GB of memory.

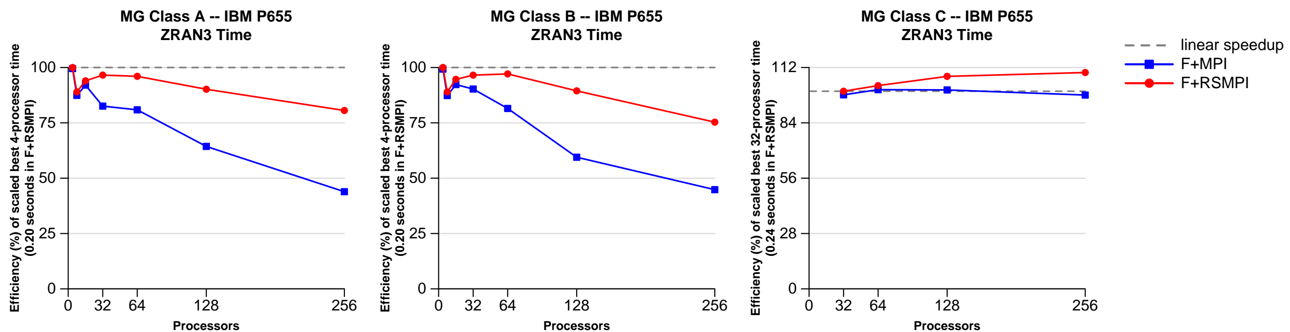

Figure 3. Efficiency graphs showing the speedup of the ZRAN3 subroutine of classes A, B, and C of the NAS MG benchmark on an IBM P655 system with 92 nodes, each containing eight 1.5 GHz power4 processors and 16 GB of memory.

higher level languages such as Chapel. We have introduced the RSMPI software to allow MPI programmers to apply custom scan and reduce conveniently using a mechanism that encapsulates the various components of the implementing parallel prefix logic. And, we have shown how custom scan and reduce outperform repeated use of built-in scan and reduce for examples from the NAS benchmarks.

The use of RSMPI to implement custom reduce and scan illustrates the application of a global-view language facility to a local-view language. One possibility for future research would be to explore whether there are other global-view language features showing great promise that might be supported in an analogous way in local-view languages.

References

[1] D. Bailey, T. Harris, W. Saphir, R. F. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report RNR-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995.

[2] R. Barriuso and A. Knies. SHMEM user's guide. Technical Report SN-2516, Cray Research Inc., May 1994.

[3] G. E. Blelloch. *Vector Models for Data Parallel Computing*. MIT Press, Cambridge, MA, 1990.

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

[5] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[6] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran performance and potential: An NPB experimental study. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2003.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, 2004.

[8] S. J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, January 2005.

[9] S. J. Deitz, B. L. Chamberlain, and L. Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1), 2002.

[10] 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.

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

[12] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Workshop on Runtime Systems for Parallel Programming*, 1999.

[13] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[14] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[15] G. Viswanathan and J. R. Larus. User-defined reductions for efficient communication in data-parallel languages. Technical Report 1293, University of Wisconsin, Madison, WI, January 1996.

[16] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.