Managing irregular remote accesses to distributed shared arrays in a bulk-synchronous parallel programming environment*

Christoph Kessler
PELAB Programming Environments Laboratory
Department of Computer Science
Linköping University, S-581 83 Linköping, Sweden
E-mail: chrke@ida.liu.se

Abstract NestStep *is a parallel programming language for the BSP (bulk-synchronous parallel) programming model. In this paper we describe the concept of distributed shared arrays in* NestStep *and its implementation on top of MPI. In particular, we describe a novel method for runtime scheduling of irregular, direct remote accesses to sections of distributed shared arrays. Our method, which is fully parallelized, uses conventional two-sided message passing and thus avoids the overhead of a standard implementation of direct remote memory access based on one-sided communication. The main prerequisite is that the given program is structured in a BSP-compliant way.*

1 Introduction

A shared memory view of a parallel computer with distributed memory can ease parallel programming considerably. Software techniques for emulating a virtual shared memory on top of a distributed memory architecture, implemented by the compiler and the runtime system, support such a shared memory view, but some remaining problems such as controlling memory consistency or processor synchronization are still under the programmer's responsibility. Sequential memory consistency is usually what the programmer wants, but maintaining this strong consistency property automatically throughout program execution causes usually a high amount of interprocessor communication. Furthermore, synchronization of processor sets is also expensive where it must be simulated by message passing. Hence, consistency and synchronicity aspects should be considered and specified together. Preferably, already the programming language itself should offer a simple but flexible construct that allows to control the degree of synchronous execution and memory consistency. This is the approach taken in the parallel programming language NestStep, which is based on the BSP model.

In this paper we describe the concept of distributed shared arrays in NestStep and its implementation. In particular we introduce a new inspector-executor approach that is suitable for irregular BSP-style computations on distributed shared arrays.

Section 2 briefly introduces the BSP model and NestStep. Replicated and distributed shared arrays are introduced in Section 3. The implementation of NestStep is described in Section 4, with a focus on distributed shared arrays in Section 4.3. Section 5 gives first results, Section 6 discusses related work, and Section 7 concludes.

2 Background

The *BSP (bulk-synchronous parallel) model* [13] structures a parallel computation of p processors into a sequence of *supersteps* that are separated by global barrier synchronization points. A superstep consists of (1) a phase of local computation of each processor, where only local variables (and locally held copies of remote variables) can be accessed, and (2) a communication phase that sends some data to the processors that may need them in the next superstep(s), and then waits for incoming messages and processes them. The separating barrier after a superstep is not necessary if it is implicitly covered (e.g., by blocking receives) in the communication phase.

The BSP model, as originally defined, does not support a shared memory; rather, the processors communicate via explicit message passing. Also, there is no support for processor subset synchronization. In order to exploit nested parallelism in programs, it must be explicitly flattened, which is generally difficult for MIMD computations. NestStep eliminates these weaknesses by providing explicit language constructs for sharing of variables and for processor subset synchronization by static and dynamic nesting of supersteps.

NestStep [8] is defined by a set of language extensions that may be added, with minor modifications, to any imperative programming language, be it procedural or object oriented. The first version of NestStep [7] was based on Java, and the runtime system of NestStep-Java was written in Java as well. However, we were so disappointed by

*This research was partially supported by a habilitation fellowship of the Deutsche Forschungsgemeinschaft (DFG).

the poor performance of Java, in particular the slow object serialization required for communication, that we decided for a redesign based on C, called NestStep-C. In the course of the redesign process, we changed the concept of distributed arrays and abolished the so-called volatile shared variables. This simplified the language design and improved its compatibility with the BSP model.

The sequential aspect of computation is inherited from the basis language. NestStep adds some new language constructs that provide shared variables and process coordination. Some restrictions on the usage of constructs of the basis language must be made for each NestStep variant. For instance, in NestStep-C, pointers are restricted; in NestStep-Java, the usage of Java threads is strongly discouraged. The NestStep extensions could as well be added to C++ or Fortran or similar imperative languages. The basis language needs not be object oriented, as parallelism is not implicit by distributed objects communicating via remote method invocation, but expressed by separate language constructs.

NestStep processes run, in general, on different machines that are coupled by the NestStep language extensions and runtime system to a virtual parallel computer. This is why we prefer to call them *processors* in the following. Each processor executes one process with the same program (SPMD), and the number of processes remains constant throughout program execution.

The NestStep processors are organized in *groups*. The processors in a group are automatically ranked (denoted by `$`) from 0 to the group size (denoted by `#`) minus one. The `main` method of a NestStep program is executed by the *root group* containing all available processors of the partition of the parallel computer the program is running on. Groups can be dynamically subdivided during program execution, following the static nesting structure of the supersteps. The NestStep feature of nestability of supersteps, denoted by the `neststep` statement, has been introduced in previous work [7, 8] but is not in the focus of this paper. Rather, it is sufficient to know that ordinary, flat supersteps are denoted in NestStep by the `step` statement

`step` *statement*

and executed by the processors of a group in a bulk-synchronous way. All processors of the same group are guaranteed to work within the same step statement (*superstep synchronicity*), which implies an implicit groupwide barrier synchronization at the beginning and the end of *statement*. Consequently, a step statement expects all processors of a group to reach the entry and exit points of that superstep, which imposes some restrictions on control flow, although for all structured control flow statements such as loops or conditional statements, the programmer can easily narrow the current group to the relevant subgroup of processors using the `neststep` statement [7, 8].

3 Sharing by replication or distribution

Beyond superstep synchronicity, the step statements also control shared memory consistency within the current group.

Variables, arrays, and objects are either *private* (local to a processor) or *shared* (local to a group). Sharing is specified by a type qualifier `sh` at declaration and (for objects) at allocation.

Shared base-type variables are *replicated* across the processors of the declaring (and allocating) group; each processor executing the declaration holds one copy. Shared arrays may be replicated or distributed; replication is the default. Pointers, if supported by the base language, can only be used as aliases for private variables and copies of shared variables in the processor's own local memory; thus, dereferencing a pointer never causes communication.

Shared arrays are stored differently from private arrays and offer additional features. A shared array is called *replicated* if each processor of the declaring group holds a copy of all elements, and *distributed* if the array is partitioned and each processor owns a partition exclusively. Like their private counterparts, shared arrays are not passed by value but by reference. For each shared array (also for the distributed ones), each processor keeps its element size, length and dimensionality at runtime in a local array descriptor. Hence, bound-checking (if required by the basis language) or `length` inspection can always be executed locally.

On entry to a step statement holds that on all processors of the same active group the private copies of a replicated shared variable (or array or heap object) have the same value (*superstep consistency*). The local copies are updated only at the end of the step statement. Note that this is a deterministic hybrid memory consistency scheme: a processor can be sure to work on its local copy exclusively within the step statement.

Computations that may access nonlocal elements of distributed shared arrays (see later) or write to replicated shared variables, arrays, or objects, must be inside a step statement or its dynamic extent. An exception is made for constant initializers.

At the end of a step statement, together with the implicit groupwide barrier, there is a groupwide *combine phase*. The (potentially) modified copies of replicated shared variables are combined and broadcast within the current group. Hence, when exiting the superstep, all processors of a group again share the same values of the shared variables. The combine function can be individually specified for each shared variable at its declaration, by an extender of the `sh` keyword [8]. This also includes the computation of global reductions (sum, maximum, bitwise operations etc.) of the modified local copies, and even prefix computations, such as prefix sums. The combine function can be also be redefined locally for each superstep. Since combining is implemented in a treelike way, parallel reductions and parallel prefix sums can be computed on-the-fly at practically no additional expense just as a byproduct of establishing superstep consistency and group synchronization.

3.1 Replicated shared arrays

For a replicated shared array, each processor of the declaring group holds copies of all elements. The combining strategy declared for the element type is applied elementwise. The declaration syntax is similar to standard Java or C arrays:

```
sh<+> int[] a;
```

declares a replicated shared array of integers where combining is by summing the corresponding element copies.

A shared array can be allocated (and initialized) either statically already at the declaration, as in

```
sh int a[4] = {1, 3, 5, 7};
```

or later (dynamically) by calling a constructor like

```
a = new_Array ( N, Type_int );
```

3.2 Distributed shared arrays

In the first NestStep design [7], distributed arrays were volatile by default, such that each array element was sequentially consistent. However, this was not compliant with the BSP model, and made the implementation less efficient. In contrast, the current language definition applies to distributed arrays the same superstep consistency as to replicated shared variables, arrays, and objects: the modifications to the array elements become globally visible at and only at the end of a superstep. Hence, the processor can be sure to work on its local copy exclusively until it reaches the end of the superstep. For concurrent updates to the same array element, the corresponding declared (or default) combine policy is applied, as in the scalar case.

The various possibilities for the distribution of arrays in NestStep are inspired by other parallel programming languages for distributed memory systems, in particular by Split-C [3].

Distributed shared arrays are declared as follows. The distribution may be either in contiguous blocks or cyclic. For instance,

```
sh int[N]</> b;
```

denotes a block-wise distribution with block size $\lceil N/p \rceil$, where p is the size of the declaring group, and

```
sh int[N]<%> a;
```

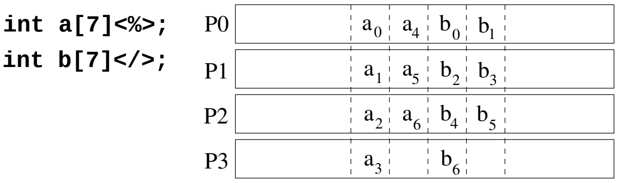

Figure 1: Cyclic and block distribution of array elements across the processors of the declaring group.

denotes a cyclic distribution, where the owner of an array element is determined by its index modulo the group size. Such distributions for a 7-element array across a 4 processor group are shown in Figure 1.

Multidimensional arrays can be distributed in up to three "leftmost" dimensions. As in Split-C [3], there is for each multidimensional array `A` of dimensionality d a statically defined dimension k, $1 \leq k \leq d$, $k \leq 3$, such that all dimensions $1, ..., k$ are distributed (all in the same manner, linearized row-major) and dimensions $k+1, ..., d$ are not, i.e. $A[i_1, ..., i_d]$ is local to the processor owning $A[i_1, ..., i_k]$. The distribution specifier (in angle brackets) is inserted between the declaration brackets of dimension k and $k+1$. For instance,

```
sh int A[4][5]<%>[7]
```

declares a distributed array of $4 \cdot 5 = 20$ pointers to local 7-element arrays that are cyclically distributed across the processors of the declaring group.

The distribution becomes part of the array's type and must match e.g. at parameter passing. For instance, it is a type error to pass a block-distributed shared array to a method expecting a cyclically distributed shared array as parameter. As NestStep-Java offers polymorphism in method specifications, the same method name could be used for several variants expecting differently distributed shared array parameters. In NestStep-C, different function names are required for different parameter array distributions.

3.2.1 Scanning local index spaces

NestStep provides parallel loops for scanning local iteration spaces in one, two and three distributed dimensions. For instance, the `forall` loop

```
forall ( i, a )
   stmt(i, a[i]);
```

scans over the entire array `a` and assigns to the private iteration counter `i` of each processor exactly those indices of `a` that are locally stored on this processor. For each processor, the loop enumerates the local elements upward-counting. The local iteration spaces may be limited additionally by the specification of a lower and upper bound and a stride:

```
void parprefix( sh int a[]</> )
{
 int *pre;     // private prefix array
 int Ndp=N/p;  // let p divide N for simplicity
 int myoffset; // prefix offset for this proc.
 sh int sum=0; // constant initializer
 int i, j = 0;
 step {
   pre = new_Array( Ndp, Type_int );
   sum = 0;
   forall ( i, a ) {  // locally counts upward
      pre[j++] = sum;
      sum += a[i];
   }
 } combine( sum<+:myoffset> );
 j = 0;
 step
   forall ( i, a )
     a[i] = pre[j++] + myoffset;
}
```

Figure 2: Computing parallel prefix sums in NestStep-C.

```
forall ( i, a, lb, ub, st )
   stmt(i, a[i]);
```

executes only every `st`th local iteration `i` between `lb` and `ub`. Downward-counting local loops are obtained by setting `lb>ub` and `st<` 0. Generalizations `forall2`, `forall3` for 2 and 3 distributed dimensions use a row-major linearization of the multidimensional iteration space following the array layout scheme.

Array syntax, as in Fortran90 and HPF, has been recently added to NestStep to denote fully parallel operations on entire arrays. For example, `a[6:18:2]` accesses the elements `a[6]`, `a[8]`, ..., `a[18]`.

For each element `a[i]` of a distributed shared array, the function `owner(a[i])` returns the ID of the owning processor. Ownership can be computed either already by the compiler or at run-time, by inspecting the local array descriptor. The boolean predicate `owned(a[i])` returns true iff the evaluating processor owns `a[i]`.

Figure 2 shows the NestStep-C implementation of a parallel prefix sums computation for a block-wise distributed shared array `a`.

3.2.2 Accessing distributed shared array elements

Processors can access only those elements that they have locally available.

A nonlocal element to be read is prefetched from its owner by a read request at the beginning of the superstep that contains the read access (more specifically, at the end of the previous superstep). Potential concurrent write accesses to that element during the current superstep, either by its owner or by any other remote processor, are not visible to this reading processor; it uses the value that was globally valid at the beginning of the current superstep. The necessary prefetch instructions can be generated automatically by the compiler where the indices of requested elements can be statically determined. Alternatively, the programmer can help the compiler with the prefetching directive `mirror`.

A write access to a nonlocal element of a distributed shared array is immediately applied to the local copy of that element. Hence, the new value could be reused locally by a writing processor, regardless of the values written simultaneously to local copies of the same array element held by other processors. At the end of the superstep containing the write access, the remote element will be updated in the groupwide combine phase according to the combining method defined for the elements of that array. Hence, the updated value will become globally visible only at the beginning of the next superstep. Where the compiler cannot determine statically which elements are to be updated at the end of a superstep, this will be done dynamically by the runtime system. The poststore directive `update` could also be applied.

Prefix combining is not permitted for distributed shared array elements.

3.2.3 Bulk mirroring and updating of distributed shared array sections

Bulk mirroring and updating of distributed shared array elements avoids the performance penalty incurred by many small access messages if entire remote regions of arrays are accessed. This is technically supported (a) by the use of array syntax in assignments, which allows to send just one access request message to the owner of each section of interest, and (b) by prefetching and poststoring directives (`mirror` and `update`), as the compiler cannot always determine statically which remote elements should be mirrored for the next superstep, although the programmer may know that in some cases.

3.2.4 Example: BSP p-way randomized Quicksort

Figure 3 shows a NestStep-C implementation of a simplified version (no oversampling) of a p-way randomized Combine-CRCW BSP Quicksort algorithm by Gerbessiotis and Valiant [4]. The algorithm makes extensive use of bulk movement of array elements, formulated in NestStep as read and write accesses to distributed shared arrays.

We are given a distributed shared array `A` that holds N numbers to be sorted. The p processors sample their data and agree on $p-1$ pivots that are stored in a replicated shared array `pivs`. Now processor i, $1 \leq i < p-1$, becomes responsible for sorting all array elements whose value is between `pivs`$[i-1]$ and `pivs`$[i]$; processor 0, for all elements smaller than `pivs[0]`, and processor $p-1$, for all elements larger than `pivs`$[p-2]$, respectively. A bad load distribution can be avoided with high probability if oversampling is applied, so that the pivots define subarrays of approximately equal size. The partitioned version of array `A` is temporarily stored in the distributed

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "NestStep.h"

/** p-way randomized Combine-CRCW BSP-Quicksort
 *  variant by Gerbessiotis/Valiant JPDC 22(1994)
 *  implemented in NestStep-C.
 */
int N=10;                // default value

/** findloc(): find largest index j in [0..n-1] with
 *  a[j]<=key<a[j+1] (where a[n] = +infty).    C's
 *  bsearch() can't be used, it requires a[j]==key.
 */
int findloc( void *key, void *a, int n, int size,
             int (*cmp)(const void *, const void *) )
{
 int j;    // for the first, use naive linear search:
 for (j=0; j<n; j++)
   if ( cmp( key, a+j*size ) <= 0)  return j;
 return n-1;
}

int flcmp( const void *a, const void *b ) // compare
{                                         // 2 floats
 if (*(float *)a < *(float *)b) return -1;
 if (*(float *)a > *(float *)b) return 1;
 return 0;
}

int main( int argc, char *argv[] )
{
  // shared variables:
  sh float *A</>, *B</>;   // block-wise distr. arrays
  sh float *size, *pivs;   // replicated shared arrays
  // private variables such as T used as destination of
  // remote read are automatically treated in a special
  // way (wrapper objects) by the compiler:
  float *T, **myslice;
  int *mysize, *Ssize, *prefsize;
  // local variables:
  int i, j, p, ndp, nmp, myndp, pivi, psumsize;
  double startwtime, endwtime;

  NestStep_init( &argc, &argv );   // sets # and $
  // N may be passed as a parameter:
  if (argc > 1) sscanf( argv[1], "%d", &N );
  p = #;
  ndp = N / p;
  nmp = N % p;
  if ($ < nmp) myndp = ndp+1;
  else         myndp = ndp;
  A = new_DArray( N, Type_float ); // dyn.alloc. A[0:N-1]
  B = new_DArray( N, Type_float ); // dyn.alloc. B[0:N-1]
  mysize = new_Array( p, Type_int );
  Ssize = new_Array( p, Type_int );
  size = new_Array( p, Type_int );
  prefsize = new_Array( p, Type_int );
  pivs = new_Array( p, Type_int );

  forall (i, A)      // useful macro supplied by NestStep
     A[i]=(double)(rand()%1000000); // create some values
                                    //       to be sorted
  myndp = DArray_Length(A); // size of my local partition

  startwtime=NestStep_time(); //here starts the algorithm

  // STAGE 1: each processor randomly selects a pivot el.
  //          and writes (EREW) it to pivs[$].
  // Pivot intervals are defined as follows:
  // I(P_0)     = ] -\infty, pivs[0] ]
  // I(P_i)     = ] pivs[i-1], pivs[i] ],  0<i<p-2
  // I(P_{p-1}) = ] pivs[p-2], \infty [

  step { /*1*/
    int pivi = DArray_index( rand()%myndp );
    pivs[$] = A[pivi];  // random index in my partition
  }
  // now the pivots are in pivs[0..p-1]

  // STAGES 2, 3, 4 form a single superstep:

  step { /*2*/
   // STAGE 2: locally sort pivots. This computation is
   // replicated, thus not work-optimal, but faster than
   // sorting on one processor only and communicating
   // the result to the others.
```

```
   qsort( pivs, p, sizeof(float), flcmp );

   // STAGE 3: local p-way partitioning in local arrays
   // A->array. I have no in-place algorithm, so I over-
   // allocate O(p*ndp) space, only npd elements of
   // which will be actually filled by the partitioning:

   myslice = (float **)malloc( p * sizeof(float *));
   myslice[0] = (float *)malloc( p * ndp *sizeof(float));
   for (i=1; i<p; i++) myslice[i] = myslice[0] + i*ndp;
   for (i=0; i<p; i++) size[i] = 0;
   forall (i, A) {  // insert my own elements in slices:
     j=findloc( &(A[i]), pivs, p, sizeof(float),flcmp);
     myslice[j][size[j]++] = A[i];
   }
   // keep the local sizes for later:
   for (i=0; i<p; i++) mysize[i] = size[i];

   // STAGE 4: Globally compute the array size of global
   // slice sizes and the prefixes of the individual
   // processor contributions. This is just done as a
   // side-effect of combining:

  } combine ( size[:]<+:prefsize[:]> ); //end superstep 2

  // This array ^^^^ notation in the optional combine
  // annotation is a hint for the compiler that the
  // shared array "size" needs not be packed elementwise,
  // thus avoiding smaller combine items.   Here, the
  // effect is marginal, as "size" is usually small.

  // After combining, size[i] holds the global size
  // of pivot interval I(i), and prefsize[i] holds the
  // accumulated length of contributions of the
  // processors P0..P{$-1} from pivot interval I(i),
  // that is, the prefix sum of all size[i] contribu-
  // tions made by processors 0...$-1.

  // STAGE 5: Write the local slices to the
  // corresponding parts in array B.

  step { /*3*/
   psumsize = 0;
   for (i=0; i<p; i++) {
    // remote write to distributed shared array B:
    B[ psumsize + prefsize[i] :
       psumsize + prefsize[i] + mysize[i] - 1 ]
      = myslice[i][ 0: mysize[i]-1 ];
    Ssize[i] = psumsize; //prefixsum size[0]+...+size[i-1]
    psumsize += size[i]; //     by replicated computation
   }
#pragma mirror( B [ Ssize[$] : Ssize[$]+size[$]-1 ] )
  } // combining includes the updates to B and prefetches

  // STAGE 6:  Allocate space for all elements in my pivot
  // interval I($), and bulk-read the myslice slices
  // from the partitioned array.

  step { /*4*/
    T = new_Array( size[$], Type_float );
    // remote read from distributed array B:
    T[ 0 : size[$]-1 ]
      = B [ Ssize[$] : Ssize[$]+size[$]-1 ];
  }

  // Stages 7 + 8 can be combined into a single superstep,
  // because Stage 7 contains only local computation.

  step { /*5*/
   // STAGE 7: Now each processor sorts its array T.

   qsort( T, /* this is recognized as a NestStep
              * array by the compiler */
          size[$], sizeof(float), flcmp );

   // STAGE 8: Now write T back to B.
   // bulk remote write to B:
   B [ Ssize[$] : Ssize[$]+size[$]-1 ]
     = T [ 0 : size[$] - 1 ]
  }

  endwtime = NestStep_Wtime();

  ... // print distributed result array and time.
  NestStep_finalize();
} // main
```

Figure 3: NestStep implementation of BSP p-way Quicksort

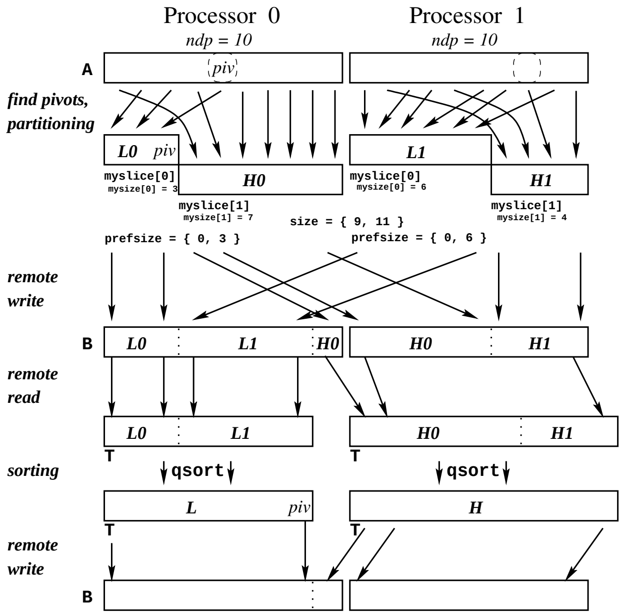

Figure 4: Illustration of the BSP randomized quicksort algorithm for $p = 2$ and $n = 20$.

target array `B`, by a remote write operation. For this operation, the processors determine the sizes and prefix sums of global sizes of the subarrays they will become responsible for. Then, they perform a remote read from `B` to local subarrays `T`. These are sorted in parallel, and then the sorted sections are copied back to `B` by a remote write operation. Figure 4 illustrates the algorithm for $p = 2$.

The example reveals that the last two communication rounds could be eliminated if the language supported irregular data distributions (here: for `B`), where the sizes of the local partitions depend on runtime data.

3.2.5 Array redistribution

Array redistribution, in particular redistribution with different distribution types, is only possible if a new array is introduced, because the distribution type is a static property of an array. Redistribution is then just a parallel assignment.

Redistribution of a distributed shared array is usually necessary at subgroup creation, where that array is accessed by a subgroup of the processors and a consistent view should be maintained during supersteps for that subgroup. This is described in detail in earlier work [7, 8].

4 Implementation with message passing

In this section, we describe an implementation of NestStep(-C) on top of a message passing layer such as MPI. We focus on those parts that handle distributed shared arrays; other details can be found in previous work [7, 8].

NestStep programs are translated by a pre-compiler to ordinary source programs in the basis language with calls to the NestStep runtime system. These files, in turn, are compiled as usual by the basis language compiler and linked with the NestStep runtime library.

4.1 Data structures of the runtime system

Group objects and group splitting The runtime system keeps on each processor a pointer `thisgroup` to a `Group` object that describes the current group of that processor. The `Group` object contains the size of the group, its initial size at group creation, the rank of that processor within the group, its initial rank at group creation, the group index, the depth in the group hierarchy tree, a superstep counter, and a pointer to the `Group` object for the parent group. Furthermore, there are fields that are used internally by the runtime system when splitting the group, such as counter arrays. There are also pointers to a list of shared variables declared by that group, to a list of references to variables that are to be combined in the next combine phase, and to a list of references to variables that are to be combined in the final combine phase at termination of that group. Finally, there is a list of hidden organizational shared variables that are used for communicating distributed shared array sections (see Section 4.3).

The details of the translation of group splitting is described elsewhere [9].

Naming schemes for addressing shared variables Generally, the same shared variable will have different relative addresses on different processors. For this reason, a system-wide unique, symbolic reference must be passed with the update value in the combine or access messages. We code names as a `Name` structure consisting of four integer components:

- the *procedure name code*, a globally unique integer hashcode identifying the declaring procedure (0 for global variables). Negative procedure names are used for internal shared variables of the runtime system.
- the *group name code*, which must be dynamically unique with respect to all predecessors and all siblings of the declaring group in the group hierarchy tree.
- a *relative name*. This is a positive integer which is given by the frontend to distinguish between different shared variables in the same activity region. Negative relative names are used internally by the runtime system.
- an *offset value*, which is used to address individual array elements and structure fields. It is -1 for scalar variables and positive for array elements and structure fields. For multidimensional arrays, the index space is flattened according to the relevant basis language policy.

Hence, equality of variables can be checked fast and easily by three or at most four integer comparisons.

Values and array objects Values are internally represented by `Value` objects. A `Value` can be an integer or floatingpoint value, or a pointer to an `Array` or `DArray` object.

`Array` objects represent replicated shared arrays. An `Array` object contains the number of elements, the element `Type`, and a pointer to the first array element in the processor's local copy.

`DArray` objects represent distributed arrays. A `DArray` object contains the number of elements, the element `Type`, the distribution type, the size of the owned array section, the global index of the first owned array element, and an `Array` object containing the owned array elements.

Shared variables Shared variables are accessed via `ShVar` objects, which are wrapper data structures for the local copies of program variables declared as shared. A `ShVar` object contains the `Name` entry identifying the variable, its `Type`, and its `Value`. The `Group` object associated with each group holds a list of `ShVar` objects containing the shared variables it has declared. Global shared variables are stored in a separate global list.

Lists of shared variables are represented by `ShVars` objects. There are methods for inserting, retrieving, and deleting `ShVar` objects from such lists. Searching for a shared variable starts in the `ShVars` list of shared variables local to the current group. If the group name code in the `Name` being searched for does not match the current group, the search method recurses to the parent group, following the path upward in the group hierarchy tree until the static scope of visibility of local variables is left (this scope is computed by the frontend). Finally, the `ShVars` list of global shared variables is searched.

Combine items A `CombineItem` object is a wrapper data structure that represents either a combine request with a value contributed to the groupwide combine phase at the end of a superstep or a commit request from another processor returning a combined value; additionally, combine items may also contain requests for remote distributed array sections and replies to such requests. These will be discussed in Section 4.3.

Combine items are designed for traveling across processor boundaries. Hence, a combine item must not contain processor-local pointers to values but the values themselves. A `CombineItem` object thus consists of the `Name` identifying the shared variable to be combined, the `Type`, a `Value` (which may include entire `Array` objects), the name of the contributing processor, and an integer characterizing the binary combine function that determines the combine policy. Combine items have an optional second `Name` entry that refers to the identifier of a private variable k, which is, for technical reasons, nevertheless registered as a `ShVar` object as well. This entry indicates the private target variable of a prefix computation for prefix combine items, and the private target array section for combine items that represent read requests to remote distributed array sections.

Combine items participating in the combine phase for the same superstep are collected in a combine list that is pointed to from the current `Group` object. A `CombineList` object has a similar structure as a `ShVars` variable list, that is, it supports dynamic insertion, retrieval, and deletion of `CombineItems`, with the exception that the combine items in a combine list are sorted by `Name` in lexicographic order.

Serialization of combine lists Combine lists can be shipped to other processors in the groupwide combine phase. For this purpose, all relevant objects (i.e., `Value`, `Array`, `DArray`, `Name`, `CombineItem` and `CombineList` objects) have serialization routines that allow to pack them into a dynamically allocated byte array. They also offer `size` routines that allow to estimate the prospective space requirement for the byte array. In Java, such routines come automatically with each `Serializable` object; in C they have been written by hand and are considerably faster.

A message is represented by a `Msg` object, which is just a dynamically allocated buffer typed `void *`. It contains the abovementioned byte array, prefixed by the byte array length, the message tag (i.e., whether it is a combine/commit message or a distributed array read/update request or reply) and the number of serialized combine items. `Msg` objects can be easily shipped across processor boundaries, for instance by using MPI routines or Java socket communication, and are deserialized to `CombineLists` on the receiver side. For this purpose, all serializable objects provide a deserialization routine as well.

Combine trees For each group g, a static *combine tree* is embedded into the given processor network. The necessary entries are collected in a `Tree` object that is pointed to by the `Group` object for the current group on each processor. The combine tree may be any kind of spanning tree, such as a d-ary tree or a binomial tree. The processors are linked such that the initial group-relative processor ranks `g.rankinit` correspond to a preorder traversal of the tree. In the `Tree` object, each processor stores the processor IDs of its parent node and of its child nodes in the combine tree.

Hardware support for treelike communication structures is clearly preferable to our software combine trees, at least for the root group. However, in the presence of runtime-data-dependent group splitting, a static preallocation of hardware links for the subgroup combine trees is no longer possible.

4.2 Combining

The combine phase at the end of a `step` consists of an upwards wave of *combine messages* (i.e., `Msg` objects tagged COMBINE) going from the leaves towards the root of the tree, followed by a downwards wave of *commit messages* (i.e., Msg objects tagged COMMIT) from the root towards the leaves. It is guaranteed that each processor participates (if necessary, by an empty contribution) in each combine phase of a group. Hence, groupwide barrier synchronization is covered by the messages of the combine phase and needs not be carried out separately.

The commit items, that are combine items communicated in the commit messages, contain the shared variable's name and its new value. Prefix commit items contain additionally an accumulated prefix value and the name of an artificial wrapper variable for the corresponding private target variable q that is to be assigned the prefix value.

The termination of a group is computed as a side-effect of combining. A special inter-subgroup combine phase is required at termination of a group, where consistency within the parent group is to be reestablished.

A detailed description of the combine and commit phases can be found in previous work [7, 8].

Note that the tree-based combining mechanism is a general strategy that supports programmable, deterministic concurrent write conflict resolution, programmable concurrent reductions, and programmable parallel prefix computations on shared variables. Nevertheless, for supersteps where the combine phase requires no prefix computations, the commit phase may be replaced by a simple native broadcast operation, which also includes the desired barrier synchronization effect. Where it is statically known for a superstep that combining is only needed for a single shared variable which is written by just one processor, the combine phase could be replaced by a (blocking) broadcast from that processor to the group, thus bypassing the combine tree structure. Where only a single variable is to be combined by reduction and it can be statically determined that *all* processors of the group contribute a new value, a collective routine for all-to-all-reduction can be used instead of general combining. Generally, static analysis of cross-processor dependences may help to replace a combine phase with point-to-point communication, which includes replacing groupwide barrier synchronization with bilateral synchronization, without compromising the superstep consistency from the programmer's point of view.

4.3 Accessing remote sections of distributed shared arrays

An update to a remote array element becomes effective at the end of the current superstep, while reading a remote array element yields the value it had at the beginning of the superstep. This implies communication of values of array elements at the boundaries of supersteps. For efficiency reasons, updates and reads to entire sections of remote array elements should be communicated together in a bulk way; the runtime system provides for this purpose the routines mirror for explicit registering of a bulk array read and update for explicit registering of a bulk array update. The programmer can help the compiler with exploiting these routines via compiler directives.

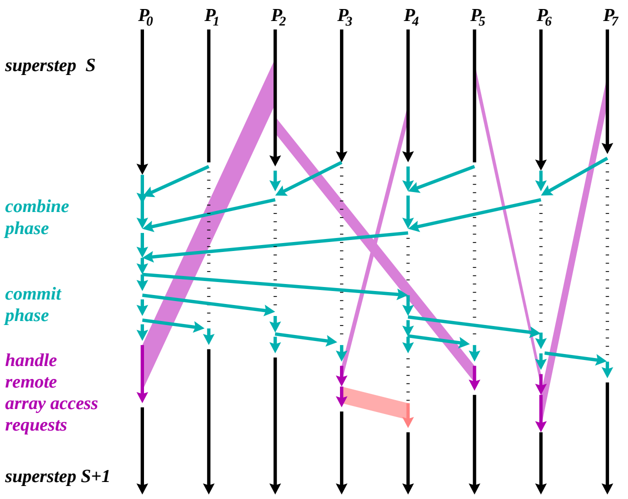

Figure 5: The numbers of remote array access requests to be expected and serviced by each processor are computed during the group-wide combine phase at the end of a superstep (here we assume a binary combine tree). The access request messages overlap in time with the combine and commit messages.

We propose a new method that overlaps the requests for reading and updating remote sections of distributed arrays with the standard combining mechanism for the replicated shared variables (as described above) and the synchronization at the end of a superstep. We show how the combining mechanism can be utilized to avoid the need of a thread-based system for one-sided communication that would otherwise be required for serving these requests.

The basic idea (see also Figure 5) is to compute at runtime, by standard combining, the number of requests that each processor will receive. Then, each processor performs that many blocking receive operations as necessary to receive and serve the requests. This partially decouples the communication of distributed array sections and of the standard combine communication, such that requests for distributed array sections can be already silently sent during the computation part of the superstep. The runtime system distinguishes between these two types of messages by inspecting the message tag in the MPI_Recv calls, such that always the right type of message can be received.

Each processor keeps in its Group object a hidden, replicated shared integer array, called N_DAmsgs[], whose (dynamically allocated) length is equal to the group size p. Entry N_DAmsgs[i] is intended to hold the number of messages with requests for reading or updating distributed array sections owned by processor i, $0 \leq i < p$, that are to be sent at the end of the current superstep. Initially, all entries in this array are set to zero. Whenever a processor j sends, during a superstep computation, a message tagged DA_REQ with a distributed array request to a remote processor i, it increments the corresponding counter N_DAmsgs[i] in its local copy. For each processor j itself, its entry N_DAmsgs[j] in its local copy of that array remains always zero. If any entry has been incremented in a superstep, the array N_DAmsgs is appended to the list of combine items for the next combine phase, with combine method Combine_IADD (integer addition). Hence, after the combine phase, array N_DAmsgs holds the global numbers of messages sent to each processor.

Now, each processor i executes N_DAmsgs[i] times MPI_Recv() with message tag DA_REQ. In the case of a read request, the processor sends the requested value back to the sender with message tag DA_REP. In the case of a write request, it performs the desired update computation with its owned array section, following the precedence rules implied by the combine method[1].

Finally, each processor executes that many times MPI_Recv as it had previously issued read requests, and stores the received array elements in the destination variable indicated in the combine item holding their values. Although this variable is, in principle, a private array, it is nevertheless registered using a ShVar wrapper object, such that it can be retrieved at this point. Finally, the counter array N_DAmsgs is zeroed again to be ready for the following superstep.

An advantage of this method is that the (parallel) histogram calculation of the numbers of outstanding request messages to be serviced by each processor is integrated into the combine phase which is to be executed anyway at the end of a superstep, hence the additional overhead caused by the runtime scheduling of the update messages is marginal.

This should be seen in contrast to the classical *inspector-executor technique* [10] that is applied to the runtime parallelization of loops with irregular array accesses in dataparallel programming environments. The inspector-executor technique applies two subsequent steps at runtime: first, the inspector (usually a sequential version of the loop restricted to address computations) determines the actual data dependencies, which imply a schedule for the actual execution of the loop and the necessary communication. Then, the executor actually executes the loop in parallel, including message passing, by following that schedule. Our method differs from this classical approach in three important aspects: First, we have a MIMD environment that is not limited to dataparallel loops. Second, our "inspector", i.e. the histogram calculation of the number of outstanding messages, fully overlaps in time with our "executor", i.e., the actual sending of access requests, by using different message tags. Finally, our "inspector" is fully parallel and works pick-a-back via the combining mechanism at virtually no extra cost.

[1] Note that prefix combining for elements of distributed shared arrays is not supported. Hence, potential combining for concurrent write updates to the same element of a distributed shared array can be done just in the order of arriving messages.

5 First results

We used the free MPICH implementation [1] of MPI as the communication layer for our prototype implementation of the NestStep-C runtime system.

For the NestStep-C version of the parprefix example (Fig. 2) we obtained the following measurements (wall clock time in seconds:

parprefix	seq	$p = 2$	$p = 3$	$p = 4$	$p = 5$
N=1000000	0.790	0.397	0.268	0.205	0.165
N=10000000	7.90	4.00	2.70	1.98	1.59

For the randomized BSP quicksort program of Figure 3 the parallel speedup is less impressive because large array sections must be shipped around in the bulk remote read and write operations, and all this communication is sequentialized on the bus network. We obtain the following figures (wall clock time in seconds):

quicksort	seq	$p = 2$	$p = 3$	$p = 4$	$p = 5$	$p = 6$
N=80000	0.641	0.393	0.428	0.437	0.391	0.375
N=120000	0.991	0.622	0.582	0.564	0.502	0.485

All measurements were taken on a network of PCs running Linux, connected by Ethernet. The NestStep application had no exclusive use of the network.

A substantial amount of performance is lost in programs with frequent synchronization because the runtime system working on top of MPI processes, which are scheduled by the local system scheduler on their machines, has no control about the scheduling mechanism. Hence, delays incurred by a processor when working on a critical path in the communication tree accumulate during the computation. We expect that this effect will be less dramatic on homogenous multiprocessors where NestStep can use the processors and the network exclusively. We also expect a considerable improvement if a scalable network can be exploited, in contrast to the bus network used above. For this reason, we are currently porting the NestStep run time system to the Cray T3E at NSC Linköping.[2]

6 Related work

In this section, we elaborate common features and differences of NestStep to widely used parallel programming environments.

The collective communication routines of MPI appear to offer an alternative to the explicit treelike combining of contributions. However, this is only applicable to very tightly coupled MIMD computations where *all* processors of a group participate in the updating of a specific

[2] However, a complete implementation and thorough evaluation of NestStep is still on my stack of unassigned master thesis projects. Interested students are cordially invited to contact me. – The author.

shared variable. In contrast, NestStep allows more general MIMD computations where not every processor of a group needs to deliver a contribution to every shared variable being written, for instance if a different control flow path within the superstep is taken that does not contain an assignment to that variable.

The one-sided communication provided by MPI-2, also referred to as direct remote memory access (DRMA), could be used as a simple way to implement a distributed shared memory. On the other hand, this requires that the programmer must explicitly care about synchronization and consistency issues, such that a BSP-style structuring of the computation is not automatically enforced.

There exist several library packages with a C interface for the BSP model. Processors interact by two-sided message passing or by one-sided communication [direct remote memory access (DRMA)]; collective communication is supported. The two most widely known BSP libraries are Oxford BSPlib [6] and Paderborn PUB [2], where the latter also supports dynamic splitting of processor groups. According to our knowledge, NestStep is the first proper programming language for the BSP model.

HPF [5] is a SIMD language and provides, at least from the programmer's point of view, a sequentially consistent shared memory. A single thread of control through the program allows to exploit collective communication routines where appropriate and supports the static and dynamic analyzability of the program. For instance, where the source and destination processor of cross-processor data dependences can be statically determined, point-to-point communication with blocking receive is usually sufficient to ensure relative synchronization and data consistency. Where senders and receivers cannot be determined statically, runtime techniques such as inspector-executor [10] may be applied. In both cases the single thread of control is an important prerequisite. HPF-2 [12] offers limited task parallelism, which is based on group splitting, but within each group there is still a single thread of control. As NestStep is a more general MIMD language, compiler techniques for HPF can be applied only in special cases.

OpenMP [11] allows general MIMD computations and provides constructs that control processor subset synchronization and consistency for individual variables. Group-wide sequential consistency is automatically enforced at explicit and implicit group-wide barriers, for example. A BSP style of programming is thus possible but not enforced or especially supported by OpenMP. Another similarity of NestStep to OpenMP is that shared variable accesses or constructs affecting parallelism and consistency, such as step statements, may occur within the dynamic but outside the static extent of the most relevant surrounding construct (e.g., a `neststep` statement). On the other hand, OpenMP is usually implemented on top of an existing (hardware or system-software emulated) shared memory platform, while the language definition of NestStep allows for the complete emulation of distributed shared memory by its compiler and run time system only.

7 Summary

We have discussed the concepts and implementation of distributed shared arrays in NestStep. In particular, we have proposed a technique that determines at runtime the structure of two-sided communication for irregular access requests to remote sections of distributed shared arrays. The necessary global parallel histogram calculation is integrated into the combine phase at the end of a superstep that is executed in any case, and hence comes at virtually no additional cost. This technique exploits the BSP structure of the parallel computation. It is an interesting alternative to one-sided communication, as the overhead of having a separate thread for servicing remote access requests on each processor can be avoided, and as multiple remote access messages to the same processor could be collected.

Acknowledgment Thanks go to Michael Gerndt for an inspiring discussion of distributed shared arrays at CPC'2000.

References

[1] ANL Argonne National Laboratories. MPICH - a portable implementation of MPI, version 1.2. www-unix.mcs.anl.gov/mpi/mpich/, 1999.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library: Design, Implementation and Performance. In *Proc. IPPSSPDP'99 IEEE Int. Parallel Processing Symp. and Symp. Parallel and Distributed Processing*, 1999.

[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 *Proc. Supercomputing'93*, Nov. 1993.

[4] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel and Distrib. Comput.*, 22:251–267, 1994.

[5] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Sci. Progr.*, 2, 1993.

[6] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: the BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.

[7] C. W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. In *Proc. PDPTA'99 Int. Conf. Parallel and Distributed Processing Technology and Applications*, volume II, pages 613–619. CSREA Press, June 28–July 1 1999. See also: Proc. 8th Workshop on Compilers for Parallel Computers CPC'00), Aussois (France), Jan. 2000, pp. 13–19.

[8] C. W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.

[9] C. W. Keßler. Parallelism and compilers. Habilitation thesis, FB IV - Informatik, University of Trier, Dec. 2000. submitted.

[10] R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of run-time support for parallel processors. In *Proc. 2nd ACM Int. Conf. Supercomputing*, pages 140–152. ACM Press, July 1988.

[11] OpenMP Architecture Review Board. OpenMP: a Proposed Industry Standard API for Shared Memory Programming. White Paper, `http://www.openmp.org/`, Oct. 1997.

[12] R. Schreiber. High Performance Fortran, Version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.

[13] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8), Aug. 1990.