Proc. of the 1999 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 28 - July 1, 1999, Las Vegas, USA. Vol. II, pp. 613–619.

NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model

Christoph W. Keßler

FB IV - Informatik, Universität Trier, 54286 Trier, Germany

Abstract. *NestStep is a parallel programming language for the BSP (bulk–synchronous–parallel) model of parallel computation. Extending the classical BSP model, NestStep supports dynamically nested parallelism by nesting of supersteps and a hierarchical processor group concept. Furthermore, NestStep adds a virtual shared memory realization in software, where memory consistency is relaxed to superstep boundaries. Distribution of shared arrays is also supported.*

A prototype for a subset of NestStep has been implemented based on Java as sequential basis language. The prototype implementation is targeted to a set of Java Virtual Machines coupled by Java socket communication to a virtual parallel computer.

Keywords: BSP model, nested parallelism, virtual shared memory, message combining, distributed arrays

1 Introduction

We describe in this paper a new parallel programming language called *NestStep*. *NestStep* is designed as a set of extensions to existing imperative programming languages like Java or C. It adds language constructs and run–time support for the explicit control of parallel program execution and sharing of program objects.

The *BSP (bulk–synchronous parallel) model*, as introduced by Valiant [16] and implemented e.g. by the Oxford BSPlib library [7] for many parallel architectures, 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 by (2). For instance, if the communication pattern in (2) is a complete exchange, a processor can proceed to the next superstep as soon as it has received a message from every other processor.

The BSP model, as originally defined, does not support a shared memory; rather, the processors communicate via explicit message passing. *NestStep* provides a software emulation of a shared memory: By default, shared scalar variables and objects are replicated across the processors. In compliance to the BSP model, sequential memory consistency is relaxed to and only to superstep boundaries. Within a superstep only the local copies of shared variables are modified; the changes are committed to all remote copies at the end of the superstep. A tree–based message combining mechanism is applied for committing the changes, in order to reduce the number of messages and to avoid hot spots in the network. As a useful side–effect, this enables on-the-fly computation of reductions and parallel–prefix computations at practically no additional expense. For space economy, *NestStep* also provides distribution of arrays in a way similar to Split-C [5].

In the BSP model there is no support for processor subset synchronization, i.e. for nesting of supersteps. Thus, programs can only exploit one–dimensional parallelism or must apply a *flattening*–transformation that converts nested parallelism to flat parallelism. However, automatic flattening by the compiler has only been achieved for SIMD parallelism, as e.g. in NESL [2]. Instead, *NestStep* introduces static and dynamic nesting of supersteps, and thus directly supports nested parallelism. There are good reasons for exploiting nested parallelism:

- "Global barrier synchronization is an inflexible mechanism for structuring parallel programs" [11].
- For very large numbers of processors, barrier–synchronizing a subset of processors is faster than synchronizing all processors.
- For parallel machines organized as a hierarchical network defining processor clusters, independently operating processor subsets may be mapped to different clusters.
- Most parallel programs exhibit a decreasing efficiency for a growing number of processors, because finer granularity means more communication. Thus it is better for overall performance to run several concurrently operating parallel program parts with coarser granularity simultaneously on different processor subsets, instead of each of them using the entire machine in a time–slicing manner.

- The communication phase for a subset of processors will perform faster as the network is less loaded. Note that e.g. for global exchange, network load grows quadratically with the number of processors. Moreover, independently operating communication phases of different processor subsets are likely to only partially overlap each other, thus better balancing network traffic over time.
- Immediate communication of updated copies of shared memory variables is only relevant for those processors that will use them in the following superstep. Processors working on a different branch of the subset hierarchy tree need not be bothered by participating in a global update of a value that they don't need in the near future and that may be invalidated and updated again before they will really need it.

A project homepage for *NestStep* is mounted at `www.informatik.uni-trier.de/~ kessler/neststep`.

2 *NestStep* Language Design

The sequential aspect of computation is inherited from the basis language. *NestStep* adds some new language constructs that provide shared variables and process coordination. In our prototype, we have designed *NestStep* as an extension of (a thread–free subset of) Java. The *NestStep* extensions could as well be added to (a subset of) C, C++, or Fortran. In particular, the basis language needs not be object–oriented: parallelism is not implicit by distributed objects communicating by remote method invocation, as in Java RMI, but expressed by separate language constructs. Admittedly, the object serialization feature of Java simplifies the implementation of sharing of arbitrary objects.

NestStep is targeted to a cluster of networked machines, defined by a hostfile, that are coupled by the *NestStep* run-time system to a virtual parallel computer. In general, on each machine runs one *NestStep* process. This is why we prefer to call these processes just *processors* throughout this paper.

NestStep processors are organized in groups. At the beginning of program execution, all processors in the cluster form a single group, the so–called *root group*, and execute the `main` method in parallel. Their number will remain constant throughout program execution (SPMD–style of parallel program execution). Groups can be dynamically subdivided during program execution, following the static nesting structure of the supersteps.

2.1 Supersteps and nested supersteps

The `step` **statement** denotes a superstep that is executed by entire *groups* of processors in a bulk–synchronous way. Thus, a `step` statement always expects all processors of the current group to arrive at this program point.

`step` *statement*

implies a group–wide barrier synchronization at the beginning and the end of *statement*[1], such that the following invariant holds: *All processors of the same group are guaranteed to work within the same* `step` *statement.* `step` also controls shared memory consistency within the current group, as will be explained in Section 2.2.

Nesting supersteps. A `step` statement with parameter(s) deactivates and splits the current group into disjoint subgroups that execute the `step` body independently of each other. The parent group is reactivated and resumes when all subgroups finished execution of the `step` body. As `steps` can be nested statically and dynamically, the group hierarchy forms a tree at any time of program execution, where the leaf groups are the currently active ones.

The first parameter of a nesting `step` specifies the number of subgroups that are to be created. Its value must be equal on all processors of the current group. An optional second parameter indicates how to determine the new subgroup for each processor. Splitting into one subgroup may make sense if only a part of the group's processors is admitted for the computation in the substep, e.g. at one–sided conditions.

`step <` k `>` *statement*

splits the current group (let its size be p) into k subgroups of size $\lceil p/k \rceil$ or $\lfloor p/k \rfloor$ each. The k subgroups are consecutively indexed from 0 to $k-1$; this subgroup index can be read in *statement* as the *group ID* `@`. Processor i, $0 \le i \le p-1$ of the split group joins subgroup $i\%k$. If $p < k$, the last $k-p$ subgroups are not executed.

In order to avoid such empty subgroups, by

`step <` k`, #>=1 >` *statement*

the programmer can specify that each subgroup should be executed by at least one processor. In that case, however, some of the subgroups will be executed serially by the same processors. Thus the programmer should not expect all subgroups to work concurrently. Also, the serial execution of some subgroups needs not necessarily be in increasing order of their group indices `@`.

In some cases a uniform splitting of the current group into equally–sized subgroups is not optimal for load balancing. In that case, the programmer can specify a weight vector to indicate the expected loads for the subgroups (this variant is adapted from PCP [3]):

`step <` k`, weight >` *statement*

Here, the weight vector `weight` must be a replicated shared array (see later) of k (or more) nonnegative floats. If the weight vector has less than k entries, the program

[1] Note that these barriers are only conceptual. In any case, the implementation tries to avoid duplicate barriers resp. combine phases.

will abort with an error message. Otherwise, the subgroup sizes will be chosen as close as possible to the weight ratios. Subgroups whose weight is 0 are not executed. All other subgroups will have at least one processor, but if there are more (k') nonzero weights than processors (p) in the split group, then the $k' - p$ subgroups with least weight will not be executed.

The group splitting mechanisms considered so far rely only on locally available information and are thus quite efficient to realize [3]. Now we consider a more powerful construct that requires group–wide coordination:

`step < k, @=` *intexpr* `>` *statement*

creates k new subgroups. Each processor evaluates its integer-valued expression *intexpr*. If this value e is in the range $[0...k-1]$, the processor joins the subgroup indexed by e. Since the subgroup to join is a priori unknown (*intexpr* may depend on run time data), the proper initialization of the subgroups (size, ranks) inherently requires another group–wide combine phase[2] that computes multiprefix–sums across the entire split group.

If for a processor the value e of *intexpr* is not in the range $[0..k-1]$, then this processor skips the execution of *statement*.

Supersteps and control flow. The usual control flow constructs like `if`, `switch`, `while`, `for`, `do`, `?:`, `&&` and `||` as well as jumps like `continue`, `break`, and `return` can be arbitrarily used within supersteps.

Nevertheless the programmer must take care that (nested) `step` statements are reached by all processors of the current group, in order to avoid deadlocks. As long as the conditions affecting control flow are *stable*, i.e. are guaranteed to evaluate to the same value on each processor of the current group, this requirement is met. Where this is not possible, e.g. where processors take different branches of an `if` statement and `steps` may occur within a branch, as in

```
if (cond)          //if stmt1 contains a step:
     stmt1();      // danger of deadlock, as
else       stmt2();//these processors don't
                   // reach the step in stmt1
```

a `step<>` statement must be used that explicitly splits the current group:[3]

```
step<2; @=(cond)?1:0>//split group into 2 subgr.
 if (@==true)stmt1();//a step in stmt1 is local
 else        stmt2();//   to the first subgroup
```

Similar constructs hold for `switch` statements. For the loops, a `step` within the loop body will not be reached by all processors of the group if some of them stop iterating earlier than others. Thus, the current group has to be narrowed to a subgroup consisting only of those processors that still iterate. A `while` loop, for instance,

```
while ( cond ) // may lead to deadlock
   stmt();     // if stmt() contains a step
```

can be rewritten using a private flag variable as follows:

```
boolean iterating = (cond);
do
   step < 1; @ = iterating?0:-1 >
      stmt();   // a step inside stmt is local
                // to the iterating subgroup
while (iterating = iterating && (cond));
```

Note that `cond` is evaluated as often and at the same place as in the previous variant. Once a processor evaluates `iterating` to 0, it never again executes an iteration. Moreover, now the iterations of the loop are separated by implicit barriers for the iterating group.

Processors can jump out of a `step` by `return`, `break`, `continue`. In these cases, the group corresponding to the target of the jump is statically known and already exists when jumping: it is an ancestor of the current group in the group hierarchy tree. On their way back through the group tree from the current group towards this ancestor group, the jumping processors cancel their membership in all groups on this path. They wait at the step containing their jump target for the other processors of the target group.

Jumps across an entire `step` or into a `step` are forbidden, since the group associated with these jump target `steps` would not yet be existing when jumping. Even if there is no `goto` in the basis language, processors jumping via `return`, `break` and `continue` may skip subsequent `step<>` statements executed by the remaining processors of their group. For these cases, the compiler either warns or recovers by applying software shadowing of the jumping processors to prevent the other processors from hanging at their next `step` statement.

Jumps within a leaf `step` (i.e. a `step` that is guaranteed to contain no other step) are harmless.

Group inspection. For each group the run time system holds on each processor belonging to it a class `Group` object. In particular, it contains the group size, the group ID, and the processor's rank within the group. The `Group` object for the current group is referenced in *NestStep* programs by `thisgroup`. `thisgroup` is initialized automatically when the group is created, updated as the group processes `step` statements, and restored to the parent group's `Group` object when the group terminates.

At entry to a `step`, the processors of the entering group are ranked consecutively from 0 to the group size minus one. This group–local processor ID can be accessed by `thisgroup.rank()` or, for convenience, just by the symbol `$`. Correspondingly, the current group's size `thisgroup.size()` is abbreviated by

[2] This combine phase may, though, be integrated by the compiler into the combine phase of the previous `step`, if k and *intexpr* do not depend on the shared values combined there.

[3] It is, in general, not advisable here to have such insertion of group splitting `steps` automatically by the compiler, since the inner `step` statements to be protected may be hidden in method calls and thus not statically recognizable. Also, paranoic insertion of group–splitting `steps` at any point of potential control flow divergence would lead to considerable inefficiency. Finally, we feel that all superstep boundaries should be explicit to the *NestStep* programmer.

the symbol #, and the group ID `thisgroup.gid()` by the symbol `@`. The `size` and `rank` fields are updated at the end of each `step`. This renumbering of the group's processors incurs only minor run-time overhead, as it is piggy-backed onto the step's combine phase. g`.depth()` determines the depth of the group g in the group hierarchy tree. The parent group of a group g can be referenced by g`.parent()`. This allows to access the `Group` object for any ancestor group in the group hierarchy tree. g`.path()` produces a string like `0/1/0/2` that represents the path from the root group to the group g in the group hierarchy tree by concatenating the `gid`s of the groups on that path. This string is helpful for debugging. It is also used by the *NestStep* run-time system for uniquely naming group–local shared variables and objects. Finally, a counter for steps executed by a group can be accessed.

Sequential parts. Statements marked sequential by

`seq` *statement*

are executed by the group leader only. The *group leader* is the processor with rank `$==0`. If the leader has already left that step, then *statement* will not be executed. Note that the `seq` statement does not imply a barrier at its beginning or end. If such a barrier is desired, `seq` has to be wrapped by a `step`:

`step seq` *statement*

Note that, if the initially leading processor leaves the group before the "regular" end of a `step` via `break`, `continue` or `return`, another processor becomes leader of the group after the current `step` and thus responsible for executing future `seq` statements.

2.2 Sharing variables, arrays, and objects

By default, basis language variables, arrays, and objects are *private*, i.e. exist once on each processor executing their declarations. *Sharing* is explicitly specified by a type qualifier `sh` at declaration and (for objects) at allocation.

Shared base–type variables and heap objects are either volatile or replicated. Shared arrays may be replicated, volatile, or distributed. In any case, replication is the default. Pointers, if existing, are discussed in Section 2.4.

By default, for a *replicated* shared variable, array, or object, one private copy exists on each processor of the group declaring (and allocating) it.

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, *volatile* if exactly one processor of the declaring group holds all the 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 run time in a local array descriptor. Thus, e.g. bound–checking (if required by the basis language) or the `length()` method can always be executed locally. For non–distributed arrays and for objects it is important to note that sharing refers to the entire object, not to partial objects or subarrays that may be addressed by pointer arithmetics. This also implies that a shared object cannot contain private fields; it is updated as an entity[4].

Sharing by replication and combining. For replicated shared variables, arrays, and objects, the `step` statement is the basic control mechanism for shared memory consistency: *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. The local copies are updated only at the end of the* `step`. 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`. In other words, we have group–wide sequential memory consistency between `step`s, and no consistency at all within `step`s. At the end of a `step`, together with the implicit barrier, there is a group–wide *combine* phase. The (potentially) modified copies of replicated shared variables are combined and broadcast within the current group according to a predefined strategy, such that all processors of a group again share the same values of the shared variables. This *combine strategy* can be individually specified for each shared variable at its declaration, by an extender of the `sh` keyword:

`sh<0> type x;` declares a variable `x` of arbitrary type `type` where the copy of the group leader (i.e., the processor with rank 0) is broadcast at the combine phase. All other local copies are ignored even if they have been written to.

`sh<?> type x;` denotes that an arbitrary updated copy is chosen and broadcast. If only one processor updates the variable in a step, this is deterministic. By

`sh<=> type x;` the programmer asserts that `x` is always assigned the same value on all processors of the declaring group; thus, combining is not necessary for `x`.

`sh<+> arithtype x;`, applicable to shared variables of arithmetic types, means that all local copies of `x` in the group are added, the sum is broadcast to all processors of the group, and then committed to the local copies. This is very helpful in all situations where a global sum is to be computed, e.g. in linear algebra applications. Here is an example:

```
sh<+> float sum;  // autom. initialized to 0.0
step
   sum = some_function( $ );
// here automatic combining of the sum copies
seq System.out.println("global sum: " + sum );
```

[4]With Java as basis language, shared objects must be `Serializable`.

There are similar reductions for global product and global bitwise AND and OR computation. `sh<foo> type x;` refers to an arbitrary associative user–defined method `type foo(type,type)` as combine function. Clearly `foo` itself should not contain references to shared variables nor steps.

Since combining is implemented in a tree–like way, prefix sums can be computed on–the–fly at practically no additional expense. In order to exploit these powerful operators, the programmer must specify an already declared private variable of the same type where the result of the prefix computation can be stored on each processor:

```
float k;
sh<+:k> float c;
```

The default sharity qualifier `sh` without an explicit combine strategy extender is equivalent to `sh<?>`.

The declared combine strategy can be overridden for individual steps by a `combine` annotation at the end of the step statement. In fact, programs will become more readable if important combinings are explicitly specified. For instance, at the end of the following step

```
sh int a, b, c;   // default combining: <?>
int k;
......
step {
   a = 23;
   b = f1( $ );
   c = f2( b );   // uses modified copy of b
}
combine ( a<=>, b<+:k>, c<+> );
```

the local copy b_i^{new} of `b` on processor i is assigned the sum $\sum_{j=1}^{p} b_j$ and the private variable `k` on processor i is assigned the prefix sum $\sum_{j=1}^{i} b_j$. Also, the copies of `c` are assigned the sum of all copies of c. The combining for `a` can be omitted for this `step`.

Replicated shared arrays. For a replicated shared array each processor of the declaring group holds copies of all elements. The array copies are combined as a whole at the end of a `step` statement. The combining strategy declared for the element type is applied element–wise to the array copies being combined:

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

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

Volatile shared variables, arrays, objects. A shared variable declared `volatile` is not replicated. Instead, it is owned exclusively by one processor of the declaring group. The owner can be specified by the programmer. It acts as a data server for this variable. Different owners may be specified for different shared volatile variables in order to balance congestion. Other processors that want to access such a variable will implicitly request its value from the owner by one–sided communication and block until the requested value has arrived. Thus, accessing such variables may be expensive. Explicit prefetching is enabled by library routines; hence, the communication delay may be padded with other computation.

Beyond its ordinary group work, the owner of a volatile shared variable has to serve the requests. Because all accesses to volatile shared variables are sequentialized, *sequential consistency* is guaranteed for them even within a superstep. Moreover, atomic operations like *fetch&add* or *atomic_add* are supported by this mechanism. The main use of volatile shared variables will be to serve as global flags or semaphores that implement some global state of computation. Thus it is important that they can be read and written to by any processor at any time, and that the result of a write access is made globally visible immediately. Although volatile shared variables do not match the BSP model directly, we believe that they are of fundamental importance for practical work.

A straightforward generalization holds for volatile shared arrays and volatile shared heap objects.

Distributed shared arrays. Only shared arrays can be distributed. Typically, large shared arrays are to be distributed to save space and to exploit locality. Distributed arrays are volatile by default, i.e. each array element resides on only one processor of the declaring group. Each access to a non–local element is automatically resolved by a blocking point–to–point communication with its owner. Hence, sequential consistency is guaranteed. However, as bulk access to remote elements will improve performance considerably, the programmer can take over control for locality and enforce explicit local buffering of elements by the `mirror` and `update` methods. In that case, memory consistency for the buffered elements is relaxed to the user's requirements.

The handling of distributed arrays in *NestStep* is partially adapted from Split-C [5]. Distribution may be in contiguous blocks

```
sh int[N]</> a;  // block distribution
```

or cyclic. Multidimensional arrays can be distributed in up to three "leftmost" dimensions. For instance,

```
sh int[4][5]<%>[7] c;   // cyclic distr.
```

distributes $4 \cdot 5 = 20$ pointers to local 7–element arrays cyclically 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 array to a method expecting a cyclically distributed array as parameter. Fortunately Java offers polymorphism in method specifications, hence the same method name could be used for several variants expecting differently distributed array parameters. In other basis languages like C, different function names are required for different parameter array distributions. `forall` loops for scanning local iteration spaces in one, two and three distributed dimensions are available in *NestStep*. Array redistribution is possible within a group by `permute()`, `scatter()` and `gather()`, or at group–splitting steps by `importArray()` and `exportArray()`. We omit the details for lack of space and refer to the following examples for illustration.

2.3 Examples

Parallel prefix computation. Reductions and prefix computations are supported already for the replicated shared variables and thus need no special communication primitives. A parallel prefix sums computation for a block–wise distributed array `a` will look as follows:

```
void parprefix( sh int[]</> a )
{
 int[] pre;     // local prefix array
 int myoffset;  // prefix offset for this processor
 sh int sum;    // automatically initialized to 0

 step {
    a.mirror( pre, a.range( $*#, ($+1)*#-1, 1 );
    for( i=1; i<pre.length(); i++)
       pre[i] += pre[i-1];
    sum = pre[pre.length()-1];
 }
 combine( sum<+:myoffset> );

 step {
    for( i=0; i<pre.length(); i++)
       pre[i] += myoffset;
    a.update( a.range( $*#, ($+1)*#-1, 1 ), pre);
 }
}
```

Parallel quicksort. The following *NestStep* routine applies a parallel recursive quicksort algorithm to a distributed array `a`.

```
void qs( sh int[]<%> a )   // a cyclically distributed
{
 // a is a distributed array of n shared integers
 sh<=> int l, e, u;
 sh<?> int pivot;
 sh float weight[2];      // replicated shared array
 int j;
 int n = a.length();

 if (n<=THRESHOLD) { seq seqsort( a ); return; }
 if (#==1)         { seqsort( a ); return; }

 while (true) // look for a good pivot element in a[]:
 {
   step {
      l = e = u = 0;
      forall(j, a, 0, #-1, 1)  // 1 local iteration
         pivot = a[j];
   } // randomly selects pivot among first elements

   // in parallel determine sizes of subarrays:
   step {
     forall (j, a)   // par. loop over owned elements
        if      (a[j]<pivot)  l++;
        else if (a[j]>pivot)  u++;
             else             e++;
   } combine ( l<+>, e<+>, u<+> ); //group-wide sum

   if (l * u > 0.17*n*n) break;   //good pivot found
 }

 // do partial presort in place in parallel:
 partition( a, pivot, l, e, u );

 // compute weight vector:
 weight[0] = max((float)(l)/(float)(l+u), 1/(float)#);
 weight[1] = max((float)(u)/(float)(l+u), 1/(float)#);
 step< 2, weight > {
    sh int[]<%> aa;
    thisgroup.importArray( aa, (@==0)? a.range(0,l,1)
                                     : a.range(l+e, u,1));
    qs( aa );

    if (@=0) thisgroup.exportArray(aa, a.range(0,l,1))
    else thisgroup.exportArray(aa, a.range(l+e,u,1));
 }
}
```

The `partition` function uses three concurrent parallel prefix computations to compute the new ranks of the array elements in the partially sorted array. The rank array entries are incremented in parallel with the respective subarray offset, and then used as permutation vector in `permute` to produce a distributed temporary array that is partially sorted. Finally, the temporary array is copied back to `a` in parallel.

2.4 Pointers

With Java as basis language, there are no pointers; references to objects are properly typed.

With C, pointer variables could be declared as shared or private. Nevertheless, the sharity declaration of a "shared pointer" variable `p` like

```
sh<+> int *p;
```

refers to the *pointee*; the pointer variable itself is always private. For shared pointers we must require that these may point to replicated shared variables, (whole) arrays, or objects only[5], and that shared pointer arithmetics is not allowed. Pointers to volatile shared variables or distributed shared arrays are not allowed. In short, pointers can be used only as aliases for processor–local data. Hence, dereferencing a pointer never causes communication.

3 Implementation

For each group, the run-time system holds a combining tree that is used to avoid hot spots in the interconnection network and support reductions and parallel prefix computations at practically no additional cost, by integrating them piggy-back in the messages that have to be sent anyway when performing write updates and barrier synchronization.

Our prototype implementation of *NestStep* for Java as basis language consists of (1) a precompiler that translates *NestStep* source code to ordinary Java source code, (2) the *NestStep* run time system, that encapsulates message passing, updating of shared variables, and group management, and (3) a driver that invokes JVM instances on remote machines listed in the HOSTFILE.

Currently, the run time system, written in Java, is operational for replicated shared variables, objects, and ar-

[5] More specifically, the shared pointer points to the local copy.

rays, and the driver is finished. Hence, simple hand–translated *NestStep* programs can now be executed.

The *NestStep* code and corresponding Java code of some example programs, together with first performance results, can be looked up at the *NestStep* WWW page.

The Java code obtained for `parprefix` (see Section 2.3) performs as follows:

`parprefix`	seq. Java	$p = 2$	$p = 4$	$p = 8$	$p = 16$
$N = 1000000$	5.4 s	7.8 s	5.6 s	3.7 s	2.2 s

These times are maximized across the processors involved (JVMs running on loaded SUN SPARC 5/10 machines), averaged over several runs. We used SUN's `javac` compiler and `java` interpreter from JDK 1.1.5 (Solaris 2.4). The implementation is optimized to avoid the overhead of parallelism if only one processor is involved. Note that the efficiency of the parallel prefix algorithm is bounded by 50 % since the parallel algorithm must perform two sweeps over the array while one sweep is sufficient for the sequential algorithm.

4 Related work

[15] provides a recent systematic survey of parallel programming languages.

Parallel Java extensions. There are several approaches to introducing HPF–like data parallelism to Java. For instance, HPJava [4] adds distributed arrays corresponding to a subset of HPF. Naturally there are some similarities between the constructs for distributed shared arrays in HPJava and these in *NestStep*. On the other hand, our framework for replicated shared variables is more flexible and powerful. Spar [10] is a similar approach for integrating array–parallel programming into Java. Spar also offers distributed arrays and a parallel forall loop, but involves a more abstract and task-parallel programming model.

Nested MIMD parallelism. An explicit hierarchical group concept as in *NestStep* is also provided e.g. in the C extensions Fork95 [8], ForkLight [9], and PCP [3].

BSP implementations. The Oxford BSPlib implementation [7], a library for Fortran and C/C++, does not support shared memory. Instead, there is one–sided communication (direct remote memory access) and various functions for message passing and global barriers. We are not aware of any high–level BSP language containing a construct like `step` that abstracts from explicit calls to barriers and message passing, although developers of BSP algorithms often use a similar notation.

[14] proposes a cost model for nested parallelism in BSP that could immediately be used with *NestStep*.

Distributed arrays. Distribution of arrays has been addressed in various languages, like Split-C [5] or HPF [13]. Unfortunately, many compiler techniques to generate and optimize communication are restricted to SIMD languages like HPF.

Virtual/distributed shared memory. Some virtual shared memory emulations are based on paging and caching. Although this allows to exploit the virtual memory management support by processor hardware and operating systems, these systems suffer from high invalidation and update traffic caused by false sharing, as the hardware is unaware of the program's variables and data structures. Also, there is little support for heterogeneous parallel systems like networks of workstations. Other approaches like Shasta [12], Rthreads [6], or this one, are object–based, i.e. sharing is in terms of data objects rather than memory pages. The Rthreads system uses a similar precompiler method of implementation.

Combining of messages. The combining trees have been inspired by hardware combining networks of some shared memory architectures like SBPRAM [1].

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, Dec. 1993.

[2] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, S. Chatterjee. Implementation of a portable nested data-parallel language. *J. of Parallel and Distributed Computing*, 21:4–14, 1994.

[3] E. D. Brooks III, B. C. Gorda, K. H. Warren. The Parallel C Preprocessor. *Scientific Programming*, 1(1):79–89, 1992.

[4] B. Carpenter, G. Zhang, G. Fox, X. Li, X. Li, Y. Wen. Towards a Java Environment for SPMD Programming. In *Proc. 4th Int. Euro-Par Conf.*, pp. 659–668. Springer LNCS 1470, 1998.

[5] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing'93*, Nov. 1993.

[6] B. Dreier, M. Zahn, T. Ungerer. The Rthreads Distributed Shared Memory System. In *Proc. 3rd Int. Conf. on Massively Parallel Computing Systems*, Apr. 1998.

[7] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, R. Bisseling. BSPlib, the BSP Programming Library. Report, see www.bsp-worldwide.org, May 1997.

[8] C. W. Keßler, H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of Parallel Programming*, 25(1):17–50, Feb. 1997.

[9] C. W. Keßler, H. Seidl. ForkLight: A Control–Synchronous Parallel Programming Language. In *Proc. High-Performance Computing and Networking*, Apr. 1999.

[10] F. Kuilman, K. van Reeuwijk, A. J. van Gemund, H. J. Sips. Code generation techniques for the task-parallel programming language Spar. In P. Fritzson, editor, *Proc. 7th Workshop on Compilers for Parallel Computers*, pp. 1–11, June 1998.

[11] W. F. McColl. Universal computing. In *Proc. 2nd Int. Euro-Par Conference*, volume 1, pp. 25–36. Springer LNCS 1123, 1996.

[12] D. J. Scales, K. Gharachorloo, C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. ASPLOS VII*, pp. 174–185, Oct. 1996.

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

[14] D. Skillicorn. miniBSP: a BSP Language and Transformation System. Technical report, Dept. of Computing and Information Sciences, Queens's University, Kingston, Canada, Oct. 22 1996. http://www.qucis.queensu.ca/home/skill/mini.ps.

[15] D. B. Skillicorn, D. Talia. Models and Languages for Parallel Computation. *ACM Surveys*, June 1998.

[16] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), Aug. 1990.