Template Based Structured Collections

Jörg Nolte, Mitsuhisa Sato, Yutaka Ishikawa
Real World Computing Partnership
Tsukuba Mitsui Bldg. 16F, 1-6-1 Takezono
Tsukuba 305-0032, Ibaraki, Japan
{jon,msato,ishikawa}@trc.rwcp.or.jp

Abstract

Collective operations on distributed data sets foster a high-level data-parallel programming style that eases many aspects of parallel programming significantly. In this paper we describe how higher-order collective operations on distributed object sets can be introduced in a structured way by means of reusable topology classes and C++ templates.

1 Introduction

Collective operations foster a high-level data-parallel programming style that hides many communication and synchronization details from application programmers. However, parallel programming languages that support collective operations are still not in widespread use and communication libraries like MPI usually restrict themselves deliberately to a few common operations, since it is notoriously hard to find suitable implementations that perform well on all use cases and all architectures. We do not want to rely on a single implementation of collections but we want to be able to control, extend and adapt collective operations on user-level by means of inheritance mechanisms and standard language features.

In this paper we describe how flexible collective operations can be introduced by means of reusable topology classes and C++ function templates. Topology classes are used to describe the *relation* between members of distributed data sets and template classes as well as function templates are applied to address such data sets collectively thus providing higher-order functions on distributed data collections.

The outline of the paper is as follows: First we give a short introduction to the Multiple Threads Template Library (MTTL) [13] on which our implementation is based. Then we discuss typical collective operation patterns and their specific characteristics. In the following sections we describe how these patterns can be implemented in a structured way without language extensions using standard C++ features only. The current implementation on top of the MTTL is discussed in detail. Performance numbers for the basic collective operations as well as combined pipelined operations are presented and discussed. Finally we compare our work to other approaches and provide some concluding insights.

2 The Multiple Threads Template Library

The MTTL is a very efficient communication and threading library on top of either the PM [14] or MPI communication layer. Amongst several other features the MTTL provides a concept for *global pointers*, multi threaded remote method invocation and global synchronisation variables.

Global pointer templates store both the network address as well as the local address of an object, such that the object can be read and written remotely using overloaded operators. Objects can be created remotely resulting in a global pointer that can also be applied for remote method invocation. Remote method invocation is implemented by means of function templates that take care of argument marshaling, communication and method invocation. A method can either be invoked synchronously or asynchronously. In case of asynchronous invocations, global pointer based synchronization variables can be used for lazy synchronization. A synchronization variable is a kind of future [12], that blocks any reading access until the variable is (remotely) initialized. These few prerequisites were sufficient to extend the MTTL with collections and efficient higher-order operations.

3 Collective Operation Patterns

Collective operations typically follow four basic patterns with possibly multiple variations (fig. 1). Pattern (a) is applicable either to initiate a parallel computation or to propagate values to all members of the collection. In the context

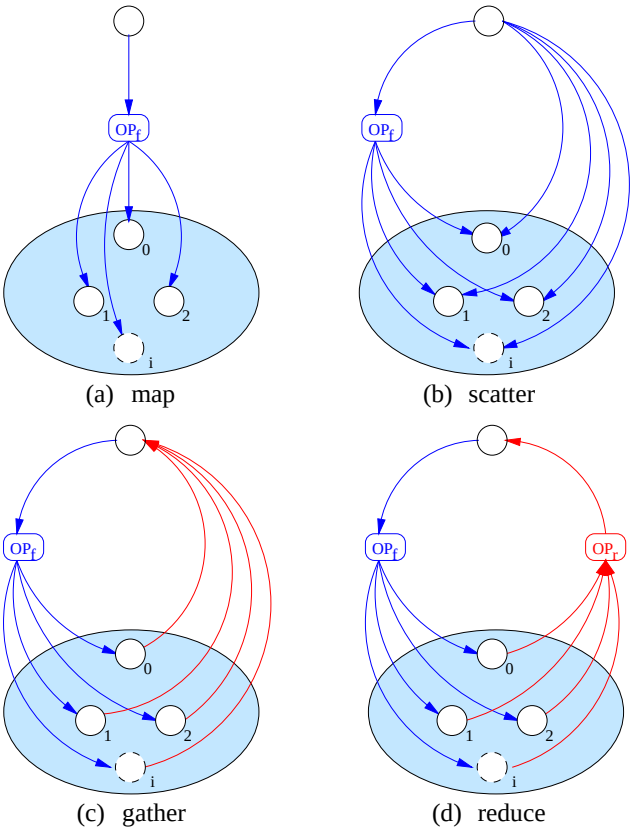

Figure 1. Collective Operation Patterns

of data parallel programming this pattern is sometimes referred to as a higher-order `map()` function [11] that maps each member `x` in a (distributed) collection to `f(x)`. Pattern (b) is a typical *scatter* pattern. A request is issued to the collection and the input is scattered amongst the members according to the rank. Conversely, pattern (c) is a *gather* pattern. A request is issued to the collection and the results are collected in one place. Pattern (d) is a global reduction pattern. Each group member delivers a result and all results are combined to a single value according to a reduction function. Such kind of operations are useful to compute global maxima, minima as well as global sums, products and the like.

A common variation to all basic patterns are selective operations that operate only on specific subsets of a collection. Furthermore a combination of reduction patterns is found in so-called `scan()` operations that compute for each member of the collection partial reductions amongst all members with either higher (*suffix* operations) or lower ranks (*prefix* operations [3]).

It should be noted that all patterns can in principle be supported by most remote method invocation systems. (a) can easily be introduced by means of *repeater* objects that act as representatives of object groups and spread a request message to all group members. (b) (c) and (d) can be implemented by passing a reference to either a remote input source or to a result object as parameter to an operation according to pattern (a). However, this will become very inefficient when object groups become larger because either a single input or a single result object has to process the messages of all group members (section 6.2).

Pattern (a) (b) and (c) can also be problematic in case it must be known when the operation is terminated or when the number of expected results is a priori unknown.

We implemented only those operations that are fundamental to all collective operation patterns. An asynchronous global `map()` operation is provided to initiate data parallel computations. The asynchronous `map()` operation is complemented with a synchronous `reduce()` operation that executes a method on all members of the collection in parallel and applies an associative and commutative function to combine all results. Both methods can be used to scatter or gather data amongst the members of a collection.

4 Collections and Topologies

The structure of a collection is defined by a topology. In principle, a collection can have any user defined topology such as n-dimensional grids, lists or trees (fig. 2).

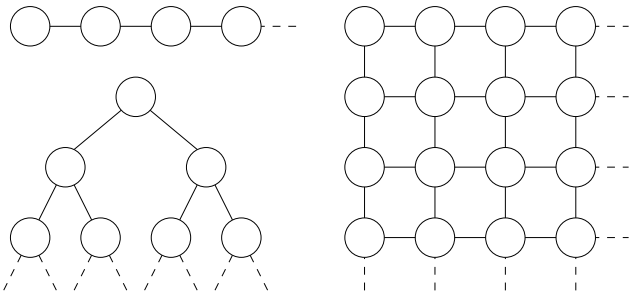

Figure 2. Topologies

Topologies are implemented as distributed linked object sets. These sets can be addressed as a whole by issuing a method to a *leader object* that represents the whole collection. Note that multiple objects of a collection might be mapped to the same computing node. The MTTL will apply local invocation mechanisms in this case. Thus the size of collections is not limited by the number of available nodes. A declaration of the form

```
CollectionOf<Sheep,BinTree> flock(64);
```

will create a distributed collection of 64 instances of class `Sheep` that is internally organized as a distributed binary tree. Note that this form of declaration is not intrusive

and thus the Sheep class does not need to be aware that its instances might belong to a collection.

Class Sheep can be any arbitrary C++ class, while BinTree is a template class that describes a distributed tree of objects of any kind. Currently all collective methods assume that a topology class defines an iterate() method that (recursively) iterates all members of a collection. In the future we will employ STL [21] iterator classes for iteration. Members are identified uniquely by means of *global pointers* (see section 2). Method calls are encoded into invocation messages and the iterator of the topology class determines the order in which these messages are passed to other members of the collection. In case of reductions all results are collected in the same order in which the invocation method has been spread. Thus, the behaviour remains deterministic and application programmers can get control over collective operations through the iterator method of the topology class. In order to determine the number of expected results, topology classes need to specify a members() method that returns the number of members that will be iterated.

In addition to the basic non-intrusive CollectionOf template, there is an intrusive GroupOf template that requires its members to be derived from a GroupMember class that reveals the structure of the collection.

```
class Sheep:
  public GroupMember<Sheep,BinTree> {
  ...
};

GroupOf<Sheep> flock(128);
```

In this case class Sheep is aware that its instances will become members of a collection. The GroupMember template requires that both the types of the collection members as well as the topology of the collection is specified. Being aware of collections and topologies is beneficial to those classes that need to determine topology related data at runtime, such as the rank of objects within the topology, the addresses of neighbouring objects in case of grid topologies and similar data.

It is generally a good idea to separate collection aware classes from other classes. Therefore it is both useful and convenient to describe a Sheep class that is aware of being in a flock by multiple inheritance as follows:

```
class FlockMember: public Sheep,
  public GroupMember<FlockMember,BinTree>
{
   // specific collective operations
   // are declared here
   ...
};
```

The FlockMember class inherits all abilities from the Sheep class and additionally knows about its group relationship. The methods of FlockMember can easily deal with conditional collective operations without any interference with the basic Sheep class.

Note that collective operations are applied in the same manner to both intrusive and non-intrusive collections, only the declarations of the collections as such differ. Furthermore, constructors of collections might have an arbitrary number of arguments that are passed to the members for initialization. When the scope of a collection object becomes invalid (or the collection object is deleted in case it has been allocated on the heap) all members of the distributed collection will also automatically be destroyed.

4.1 Addressing of Individual Instances

The basic framework for collective operations is based on dynamic data structures and global pointers instead of index spaces as e.g. in Promoter [2].

However, for convenience reasons both the GroupOf as well as the CollectionOf class provide overloaded subscript-operators such that members within a collection can be randomly addressed like array members.

```
GroupOf<Sheep> flock(128);
int result;

invoke(result,flock[5],&Sheep::weight);
```

The invoke() function template is the MTTL way to invoke a remote method on another processor. The flock[5] expression evaluates to a global pointer to a Sheep object on which the weight() method of the Sheep class is invoked. Note that the actual type arguments to the invoke function template are automatically determined by the C++ compiler's template matching mechanism, that "fills in" the type parameters using the types of the actual arguments. Therefore type checking can also be performed at compile time.

Since each member of a collection is uniquely identified by a global pointer, it is very easy to implement mapping classes that map any type of user defined addressing (let it be indices, name strings or whatever is appropriate) to the corresponding global pointer to the object. Thus addressing of individual members is kept as flexible as the construction of collections themselves and any style of individual adressing is possible.

4.2 Collective Methods

A collection object represents all members of a collection and thus each method that is executed on the collection object is in fact executed in parallel on each member of the collection. The most simple collective method that can be executed on collections is the map method.

`map()` invokes a specified method without return values in parallel on all members of the collection. When `coll` is a collection of member type `class` and `class::method` is a `void` member function of `class`, `map()` invokes `class::method` on all members of `coll` in the order defined by the iterator of the collections topology.

```
coll.map(&class::method,args);
```

Consider a class `Sheep` provides a `feed()` method that assigns each `Sheep` instance an amount of food. Then `map` can be applied to assign each member of the collection the same amount of food:

```
void Sheep::feed(int amount) { ... }

GroupOf<Sheep> flock(128);
// invoke feed(12) on all members
flock.map(&Sheep::feed,12);
```

Note that `map` executes asynchronously in parallel with the caller. To ensure that the operation is actually completed global reductions can be applied. Multiple `map` calls issued to the same collection are implicitly pipelined and thus can additionally speed-up parallel computations significantly.

4.3 Global Reductions

A global reduction is a synchronous data-parallel operation that first executes a specified method on all members of a collection in parallel and then combines the results by a binary reduction function.

When `coll` is a collection of member type `class` and `class::method` is a member function of `class` returning a result of type `T` then a global reduction is implemented as a collective method `reduce()` on `coll` that first executes the member function `class::method` on all members of the collection `coll` in the order defined by the iterator of the collection's topology and then combines the results by means of the associative and commutative binary function `red()` iteratively applied in the same order as defined by the iterator of the collection's topology.

```
T r = coll.reduce(&red,&class::method,args);
```

Consider a `Sheep` class which declares a `wool()` method that returns the amount of wool of an individual sheep. The `reduce()`-statement in the following piece of code then calculates the weight of the wool of a whole flock of sheep in parallel:

```
int Sheep::wool() { return my_amount; }
int add(int a, int b) { return a + b; }

GroupOf<Sheep> flock(128);
int sum = flock.reduce(&add,&Sheep::wool);
```

When the `reduce()` method is invoked on the `flock` object with a pointer to the `wool()` method of `Sheep`, `wool()` is executed on each member of the collection. The pointer to the binary `add()` function is passed to specify a reduction function for the results.

If we wanted to find out the maximum amount of wool of an individual sheep in the flock we would just need to slightly change the code and specify a `max()` function for reduction.

```
int max(int a, int b)
  { return (a > b) ? a : b; }

GroupOf<Sheep> flock(128);
int res = flock.reduce(&max,&Sheep::wool);
```

4.4 Conditional Collective Operations

We did not implement conditional collective operations because they can be implemented easily by means of *conditional wrappers* that evaluate whether a collective method should be applied to a specific instance. For instance, when we would like to sum up the wool of only those sheep of the flock whose weight is above a certain threshold we can write the following:

```
int Sheep::shear(int weight)
{
   if (my_weight >= weight)
     return wool();
   else return 0;
}

GroupOf<Sheep> flock(128);
int sum; const int MIN = 20; // min. weight

sum = flock.reduce(&add,&Sheep::shear,MIN);
```

These wrapper methods in fact allow dynamic subsetting of the collection. However, when we intend to perform many methods on the same subset of a collection, we might first create a cloned set containing only those instances that meet the specified requirements. This is a straightforward task, since collections are based on dynamic distributed data structures. Topology classes usually provide a `join()` method to add new members and thus all those members that meet the requirements can create a copy of themselves and `join()` these copies to a new collection.

4.5 Passing Parameters

All parameters to collective operations are passed by value and the size of the parameters must be known at compile time. Arrays cannot be passed directly because the implementation of the basic `invoke` function template of the MTTL expects that parameters can be assigned to instances of the same type when it encodes actual invocations into messages (*marshaling*). Arrays cannot be automatically assigned to other arrays but objects containing arrays can be

assigned to each other and thus, passing arrays either requires to use such wrapper classes or apply global pointers directly.

Global pointers are defined in the basic MTTL and allow remote memory access. Anyway, global pointers should only be applied with great care in combination with collective operations, because they can cause implicit sequentialization of parallel operations. Consider a `Sheep` class that wants to provide a collective method to assign each sheep its individual portion of food (essentially a scatter operation according to the rank in the collection). A straight forward implementation might look like this:

```
void Sheep::feed(GlobalPtr<int> food)
{
  // caution: remote access!
  my_weight += food[rank()];
}

GroupOf<Sheep> flock(128);
int food[128];

// prepare some food for each sheep
...
food[5] = 18;
...
// finally scatter the food
flock.map(&Sheep::feed,(GlobalPtr<int>)food);
```

Each `Sheep` instance accesses its individual amount of food by accessing a remote array located at the node that initiated the `map` operation. Although the `feed` method as such is executed in parallel, accessing the remote `food` array implicitly sequentializes the operation because all accesses to that array need to be handled on a single node.

Provided the input data is not excessively large, even on low latency networks the (parallel) copy overhead of passing an array by value is often faster as a global pointer access.

Note that the template matching mechanism also allows to pass pointers to local data to remote methods. This is inherently dangerous because the pointer usually does not point to a defined location at the destination site. However, in a homogeneous SPMD program all data objects within file scope (file global and/or program global variables) reside at the same logical address on each node. Therefore, when applied with great care, collective operations can also be used to globally initialize or reduce file scope data on each node.

5 Implementation of Collections

Due to space constraints we mainly discuss the `GroupOf` implementation, the implementation of `CollectionOf` follows similar design principles. Both the `CollectionOf` and the `GroupOf` class are templates that themselves intensively rely on member function templates. Unlike ordinary function templates member function templates are defined within the scope of a class declaration and thus can be invoked like a member function using either the `.` operator or the `->` member selection operator as shown in the examples throughout the paper.

The `GroupOf` class provides several overloaded constructors, that are implemented as member function templates. The compiler will automatically choose the correct implementation when a `GroupOf` instance is declared. Furthermore the corresponding constructor template will be automatically "filled in" with the types of the actual arguments of the constructor.

```
template<class T>
class GroupOf: public T {
 public:
  GroupOf(int size)
     { ... }
  template <class ARG>
  GroupOf(ARG a0,int size)
     { ... }
  template <class ARG0, class ARG1,...>
  GroupOf(ARG0 a0, ARG1 a1,...,int size)
     { ... }
  GlobalPtr<T> operator[](int i)
     { return node[i]; }
  ...
};
```

Each constructor will create `size` objects and join them according to a `join` method provided by the topology class. Note that the creation is still sequential and that the mapping of objects to computing nodes follows a fairly simple wrap around strategy. We intend to implement the mapping by means of mapping classes in the future such that both the mapping of objects to nodes as well as the alignment of objects within different collections can be controlled effectively.

The `GroupOf` class is derived from the class of the members it represents and thus is in fact simultaneously the initial member of a collection and the representative of the whole collection. The `GroupOf` class expects that all members of the collection are aware that they are part of a collection. Therefore a `GroupOf` object is implicitly itself a `GroupMember` by inheritance because template parameter `T` (the member type) is expected to be derived from class `GroupMember`.

5.1 Implementation of Collective Operations

The `map()` as well as `reduce()` methods are implemented as member function templates of the `GroupMember` class. `GroupMember` is itself a template and its template arguments describe the member type of the objects as well as the actual topology used. Note that topology classes

are expected to be template classes themselves which is syntactically expressed by declaring the formal template parameter `Topology` as a template.

```
template<class T,
   template<class T1> class Topology>
class GroupMember: public Topology<T>
{
 public:
 ...
 template<class Res,class ARG0,...>
 Res reduce(Res (*red)(Res,Res),
   Res (T::*func)(ARG0,...),ARG0 a0,...)
 {
   // 1. sub-results
   Sync<Res> results[members()];

   // 2. spread method
   for (int i=0; i<members(); i++) {
     // iterate members and invoke async.
     GlobalPtr<T> next = iterate();
     ainvoke(results[i], next ...);
   }

   // 3. local call
   Res tmp = (((T*)this)->*func)(a0,...);

   // 4. and 5. combine results
   for (int j=0; j<members(); j++) {
     // block implicitly,
     // when results[j]
     // is not yet defined!
     tmp = (*red)(tmp,*results[j]);
   }
   return tmp; // 6. return result
  }
}
```

The various `reduce` member function templates differ only in the number of arguments to pass. First an array of result variables is declared to receive all expected subresults[1](step 1). The number of results is determined by a `members()` method, which is also defined by the `Topology` class. Note that the results are declared as a kind of futures [12] using the `Sync` class of MTTL, that blocks any reading access until the result is defined. Therefore we can use lazy synchronization when we reduce the results in step 4/5.

In step 2 the collective method is spread asynchronously to all members of the collection which will in turn recursively continue this propagation process in parallel. The members are determined using an `iterate()` method that has to be defined by the `Topology` class. This iterator therefore controls method propagation for all collective methods. After the method has been propagated to other members of the collection, it is executed locally on the local instance in parallel with the method invoked on the other collection members. The degree of achievable parallelism depends on the topology of the collection and the corresponding iterator method. Tree topologies offer naturally the highest degree of inherent parallelity while other topologies such as lists or meshes might need more sophisticated iterators to achieve similar parallel performance.

Finally all results are iteratively reduced according to the specified reduction function. The computed result is returned to the basic MTTL environment that will implicitly reply the result to the invoker (step 6). The implementation of the `map()` members function templates is similar and straightforward. Since no results have to be reduced only step 2 and 3 have need to be carried out.

[1]We used a g++ specific form of array declaration here, since it usually results in faster code. Standard C++ only accepts constant expressions for array dimensions and thus will require a `new` operation for array allocation.

6 Performance

We measured the performance of our collective operations on the RWC PC Cluster II consisting of 128 Pentium Pro nodes (200MHz CPU) interconnected by a 160MByte/sec Myrinet network running the Score [14] system software on top of a Linux System.

We used a benchmark class with a `BalancedTree` topology that implements balanced n-ary trees. We have chosen a balanced binary tree for most of our measurements. This topology is not optimal as several researchers [10, 19] already have pointed out. However, since we basically want to show that our basic runtime mechanisms are scalable and efficient, we can neglect this fact.

The objects of the collection have been mapped to the computing nodes using a straightforward wrap around strategy. All benchmarks have been run 100 times in a loop to ensure, that the benchmark code was mostly in the cache and that the benchmark times were long enough to get a meaningful measurement using the standard `gettimeofday()` system call. The resulting times have been divided then by the number of runs.

6.1 Asynchronous Collective Operations

Note that all asynchronous collective `map()` operations could not be measured directly, because there is no way to determine the termination of an asynchronous collective operation without performing some synchronization code. Therefore we measured the corresponding `reduce` operations and calculated the time of the corresponding `map()` operations as follows:

We first measured a synchronous global reduction without arguments and divided the time by two to get a realistic value for the corresponding basic `map()` operation (that requires half the amount of messages of a reduction). Note

that this value is slightly higher as the true elapsed time, because some reduction code as well as lazy synchronization code is executed locally, that is not executed using the `map()` operation. This basic correction time then has been subtracted from all collective operations with parameters to calculate the corresponding `map()` times. Because of the synchronization all times do not reveal any pipelining effect.

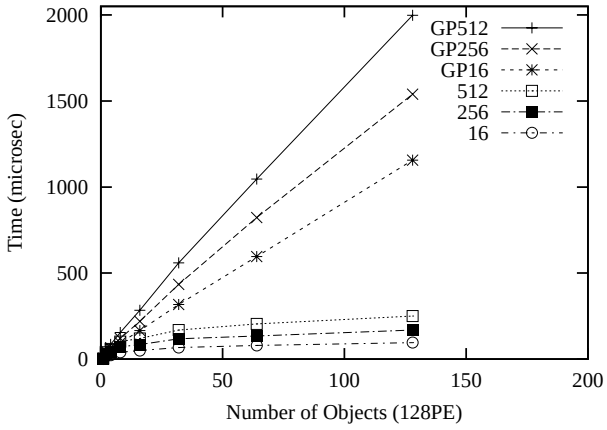

Figure 3. Copy vs. global pointer access

Figure 3 shows the results for `map()` operations with various argument sizes (16, 256, 512 bytes) and `map()` operations using global pointers (GP16, GP256, GP512) to read their arguments in a single remote access. All `map` operations which copy their input parameters recursively scale very well and the graph shows the expected logarithmic shape for a binary tree topology. In contrast, the sequentializing effect of global pointer access is clearly visible and those operations do not scale well at all.

6.2 Global Reductions

We measured parallel global sums against manual global sums. In the parallel case we executed a reduction using a simple `count()` method.

```
int Sheep::count() { return 1; }
...
int res = flock.reduce(&add,&Sheep::count);
```

In contrast the manual reduction was performed passing a global pointer to a `Sum` object to all members of the collection by means of a `map()` operation.

```
int Sheep::inc(GlobalPtr<Sum> c)
    { ainvoke(c,&Sum::inc); }
...
Sum sum(...);
flock.map(&Sheep::inc,(GlobalPtr<Sum>)&sum);
int result = sum; // lazy synchr.
```

The `Sum` class has been implemented as a future that blocks reading access until all expected results have arrived. Figure 4 shows the results of the measurements using a 4-ary tree topology. The parallel reduction scales very well. In contrast

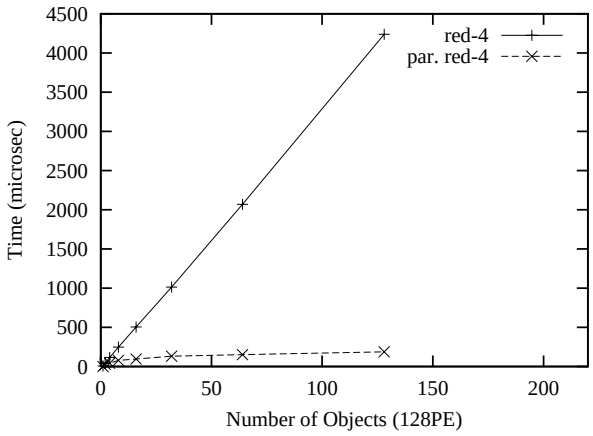

Figure 4. Parallel and manual global reductions

in case of the manual reduction the sequentializing effect of remote method invocation of the `Sum` object is clearly visible.

For comparison with a standard communication library we measured our reductions against collective operations of MPI (fig. 5) using both a binary tree (par. red-2) as well as a 4-ary tree (par. red-4) to study the impact of different tree topologies (which are both not optimal for software multicasts [10]). A `MPI_bcast()` followed by a `MPI_Reduce()` (mpi-bc-red) is closest to our `reduce()` semantics. The MPI implementation we used is MPICH-

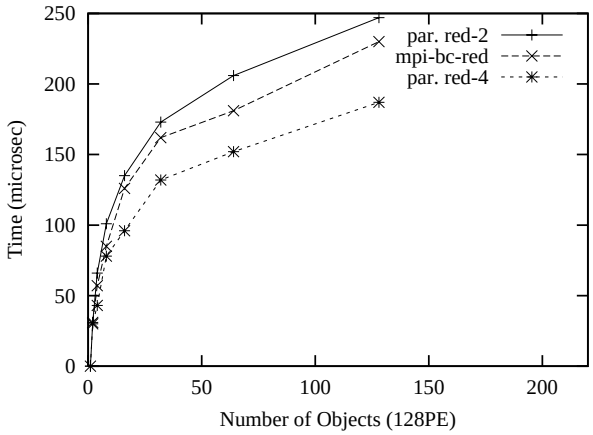

Figure 5. reduce() vs. MPICH-PM/CLUMP

PM/CLUMP [22, 18] which is a very efficient port of MPICH for Myrinet networks. Figure 5 shows that even our very first implementation is by all means in the competitive range. In case of the 4-ary tree topology our implementation shows even better performance than the corresponding MPI implementation.

6.3 Scan Operations

In a scan operation all members of a collection compute a partial reduction amongst all those members either with lower-equal ranks (*prefix*) or higher-equal ranks (*suffix*) [11]. Scan operations are not only very important for many numerical algorithms [3], but they are suitable to study pipelining effects and the impact of topologies on collective operations. We measured therefore several prefix calculations over various balanced tree topologies.

First we implemented a partial reduction by means of a conditional method (section 4.4) such that we could calculate the prefix sum for each rank by means of a `reduce()` operation. We enumerated all ranks iteratively in a loop and performed for each rank the corresponding partial reduction. Thus each partial reduction is executed as a parallel collective operation but no pipelining effects could take place between adjacent reductions.

In the next step we introduced a collective operation that computed all partial reductions in parallel (a synchronous global reduction of partial reductions) causing adjacent partial reductions to be pipelined. Figure 6 shows the results

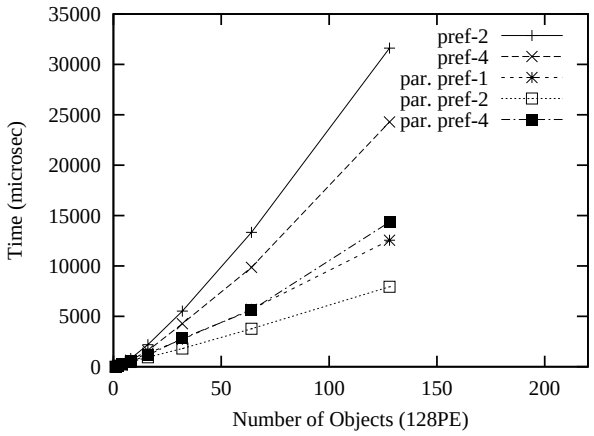

Figure 6. Prefix calculations

for various tree topologies including a linear list. For all graphs `pref-i i` denotes the arity of the corresponding tree topology. As expected, the pipelined operations (`par. pref-i`) show superior performance in comparison to the iteratively executed prefix operations (`pref-i`). We do not show the graph for `pref-1` (the iteratively computed prefix for a linear list topology) since it disturbs the scale of the axes due to its extremely bad performance.

However, in case of `par. pref-1` the simple linear list is in fact competitive to tree topologies and this shows clearly, how pipelining can contribute to parallel performance. It is therefore no surprise that the balanced binary tree shows the best performance. This topology has both an exponential operation propagation rate as well as very short node blocking times. The latter enables very good pipelining, because each node only needs to propagate to two children nodes and thus it is quickly able to proceed with the next operation while the previous operation proceeds in parallel.

7 Conclusion

Collections and collective operations are neither new nor did we invent the notion of topologies as such. In fact our work has been strongly inspired by the collection concept of pC++ [5] and ICC++ [9], the *communities* in Ocore [16], *groups* in HPC++ [1] and – last but not least – the topology concept of Promoter [2].

In Promoter topologies can be defined as constraints over n-dimensional index spaces. These topologies are both used to describe distributed variables (such as matrices) as well as coordinated communications between those variables. Data parallel expressions and statements simultaneously involve parallel computations, (implicit) communications as well as reductions. Typical numerical problems can therefore be expressed with a minimum of programming effort on a very high level of abstraction. However, although most data-parallel approaches support regular indexed based (distributed) array data structures very well it is usually not easy to deal with distributed dynamic data.

We reversed that scheme and based our collections inherently on dynamic data structures similar to the multicast topologies of ARTS [6]. Therefore existing collections can potentially be changed dynamically at run time and collections might choose any user defined addressing scheme to access member objects. The latter is a major difference to approaches like Amelia [20] or Chaos++ [8] that inherently rely on indexed based addressing of collection members.

Like in ARTS we provide effective control over group communication mechanisms through a simple iterator method that controls both the propagation of collective methods as well as the reduction of the computed results [17]. Therefore existing topologies can be reused for globally synchronized operations and specialized collective operation patterns can also be implemented with modest effort.

While many approaches like ICC++ [9] and CC++ [7] as well as ARTS are based on some language extensions,

we have shown that it is in principle possible to provide a high-level data-parallel programming abstraction without any language extension. Function templates and inheritance mechanisms of C++ are sufficient to implement structured collections and flexible as well as efficient collective operations. The global pointer concept found in MTTL (as well as in HPC++, too) eased the implementation of such structures greatly.

However, there are limits to this approach. The compiling and the debugging of complex template libraries can be a true nuisance, especially since not all compilers are able to compile all template classes correctly. Fine grained parallel computing like in ICC++ [9] or Nesl [4] is possible in principle, but we do not expect satisfactory performance results since we can neither apply aggressive compiler optimizations nor optimizing program transformations [15] with a pure library implementation. On the other hand our approach is well suited to medium to coarse grained parallel algorithms and does not require a major reimplementation of legacy code in a new data-parallel paradigm.

References

[1] P. Beckman, D. Gannon, and E. Johnson. Portable Parallel Programming in HPC++. Technical report, Department of Computer Science, Indiana University , Bloomington, IN 47401.

[2] M. Besch, H. Bi, P. Enskonatus, G. Heber, and M. Wilhelmi. High-Level Data Parallel Programming in PROMOTER. In *Proc. Second International Workschop on High-level Parallel Programming Models and Supportive Environments HIPS'97*, Geneva, Switzerland, April 1997. IEEE-CS Press.

[3] G. E. Blelloch. Prefix Sums and Their Applikations. Technical Report CMU-CS-90-190, Carnegie Mellon University, Pittsburgh, PA 15213, 1990.

[4] G. E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[5] F. Bordin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), Fall 1993.

[6] L. Büttner, J. Nolte, and W. Schröder-Preikschat. ARTS of PEACE – A High-Performance Middleware Layer for Parallel and Distributed Computing. *Journal of Parallel and Distributed Computing*, 59(2):155–179, Nov 1999.

[7] K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[8] C. Chang, A. Sussman, and J. Salz. CHAOS++. In *Parallel Programming using C++*, pages 131–174. MIT Press, 1996.

[9] A. Chien, U. Reddy, J.Plevyak, and J. Dolby. ICC++ – A C++ Dialect for High Performance Parallel Computing. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software, ISOTAS'96*, Kanazawa, Japan, March 1996. Springer.

[10] J. Cordsen, H. W. Pohl, and W. Schröder-Preikschat. Performance Considerations in Software Multicasts. In *Proceedings of the 11th ACM International Conference on Supercomputing (ICS '97)*, pages 213–220. ACM Inc., July 1997.

[11] S. Gorlatch. Towards Formally-Based Design of Message Passing Programs. In *Proceedings of the 4th IPPS/SDP International Workshop on High-Level Parallel Programming Models and Supportive Environments,IPPS/SDP99, San Juan, Puerto Rico*, April 1999.

[12] R. H. J. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985.

[13] Y. Ishikawa. Multiple threads template library. Technical Report TR-96-012, Real World Computing Partnership, 1996.

[14] Y. Ishikawa, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, F. O'Carroll, and H. Harada. RWC PC Cluster II and SCore Cluster System Software – High Performance Linux Cluster. In *Proceedings of the 5th Annual Linux Expo*, pages 55 – 62, 1999.

[15] G. Keller and M. M. T. Chakravarty. On the Distributed Implementation of Aggregate Data Structures by Program Transformation. In *Proceedings of the 4th IPPS/SDP International Workshop on High-Level Parallel Programming Models and Supportive Environments,IPPS/SDP99, San Juan, Puerto Rico*, April 1999.

[16] H. Konaka, M. Maeda, Y. Ishikawa, T. Tomokiyo, and A. Hori. Community in Massively Parallel Object-based Language Ocore. In *Proc. Intl. EUROSIM Conf. Massively Parallel Processing Applications and Development*, pages 305–312. Elsevier Science B.V., 1994.

[17] J. Nolte. Flexible Collective Operations for Distributed Object Groups. In *Proceedings of the 4th IPPS/SDP International Workshop on High-Level Parallel Programming Models and Supportive Environments,IPPS/SDP99, San Juan, Puerto Rico*, April 1999.

[18] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *International Conference on Supercomputing '98*, pages 243–250, July 1998.

[19] J.-Y. L. Park, H.-A. Choi, N. Nupairoj, and L. Ni. Construction of Optimal Multicast Trees Based on the Parameterized Communication Model. Technical Report MSU-CPS-ACS-109, Michigan State University, 1996.

[20] T. J. Sheffler. The Amelia Vector Template Library. In *Parallel Programming using C++*, pages 43–90. MIT Press, 1996.

[21] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett Packard Laboratories, 1994 revised 1995.

[22] T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. H. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. In *Parallel and Distributed Processing – IPPS/SPDP'99 Workshops*, volume 1586 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1999.