Flexible Collective Operations for Distributed Object Groups

Jörg Nolte

GMD FIRST
Rudower Chaussee 5
D-12489 Berlin, Germany
jon@first.gmd.de

Abstract. Collective operations on multiple distributed objects are a powerful means to coordinate parallel computations. In this paper we present an inheritance based approach to implement parallel collective operations on distributed object groups. Object groups are described as reusable application-specific classes that coordinate both operation propagation to group members as well as the global collection (reduction) of the computed results. Thus collective operations can be controlled by applications using language-level inheritance mechanisms. Existing group classes as well as global coordination patterns can therefore effectively be reused.

1 Introduction

Collective operations are a powerful means to implement coordinated operations on distributed data-sets as well as synchronized reductions of multiple computed results. In this paper we present an inheritance based approach to implement parallel operations on distributed data-sets as multicasts based on point-to-point communication. We want to be able to control, extend and adapt global operations on user-level by means of standard inheritance mechanisms. Multicast groups are therefore described as reusable application-specific topology classes that coordinate both the spreading of multicast messages and the collection (reduction) of the computed results. Thus global operations are controllable through applications and existing communication topologies can potentially be reused for global operations.

Several common multicast patterns are supported and a generator tool generates suitable client and server-stubs for group communications from annotated C++-classes. Our system was initially designed as an extension to the remote object invocation (ROI) [9] system of the PEACE[12] operating system family mainly targeting tasks like parallel signal propagations, name server queries and page-(in)validations in our VOTE[3] VSM system. Consequently our work has a system biased scope like early work on fragmented objects in SOS[13] with language-level support (FOG[5]) for group operations. Nevertheless, medium to coarse grained data-parallel computations can also be expressed with modest effort and also executed efficiently.

We neither rely on complex compiler technology for data-parallel languages nor do we imply applications to be structured as SPMD programs. Therefore we are able to support a parallel server paradigm that allows multiple clients to access available parallel services running either on parallel computers or homogeneous workstation clusters. This scenario is specifically useful to increase multi-user performance when developers or scientists work on related topics and use parallel computers as domain-specific parallel servers.

In the following sections we discuss the design of our multicast framework. Furthermore, we describe our language-level support and how it is compiled. Finally we present performance results and compare our work with other approaches.

2 Collective Operation Patterns

Collective operations typically follow three basic patterns with possibly multiple variations (fig. 1).

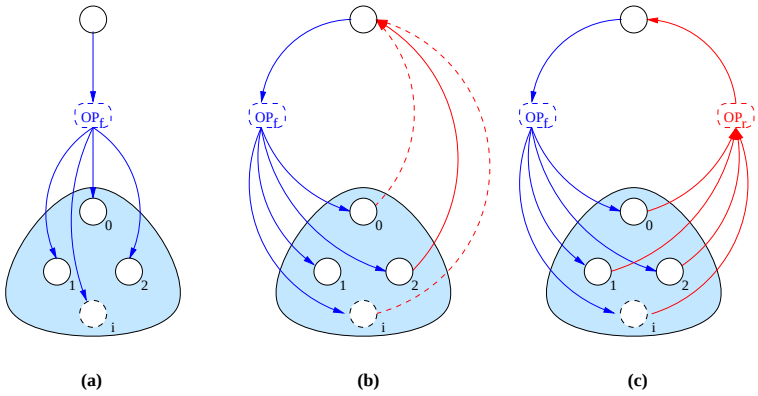

Fig. 1. Collective Operation Patterns

Pattern (a) is applicable to initiate a parallel computation or to propagate values to all members of the group. Pattern (b) is a typical search pattern. A request is issued to the group and only some specific objects deliver a result. Pattern (c) is a global reduction pattern. All group members deliver a result that is reduced according to a reduction function. Such kind of operations are useful to compute global maxima, minima as well as global sums and the like.

It should be noted that pattern (a) and (b) can in principle be supported by most remote object 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) can be implemented by passing a reference to a remote result object to an operation according to pattern (a). (c) can be

implemented in a similar way. However, (c) will become very inefficient when object groups become larger because a single result object has to process the result messages of all group members.

Pattern (a) and (b) can also be problematic if it must be known when the operation is terminated or the number of expected results is a priori unknown. Considering these potential problems we extended the PEACE ROI system with flexible collective operations based on multicast mechanisms.

2.1 Multicast Groups

In our ROI system multicast groups are implemented as distributed linked object sets. These sets can be addressed as a whole by issuing a multicast operation to a *group leader* representing the group. For example, if the group is organized as a tree (fig. 2), the root acts as the group leader and multicast operations issued to the root are executed on each object within the group. In principle, a multicast group can have any user defined topology such as n-dimensional grids, lists or trees.

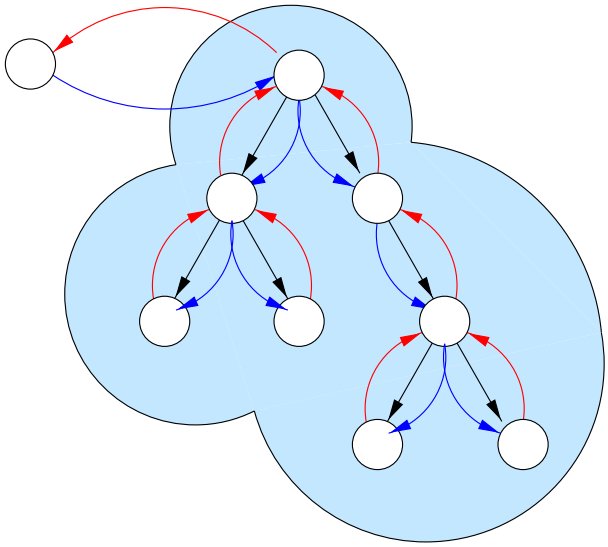

Fig. 2. A Multicast Topology

These topologies are described through reusable C++ topology classes. Members of a multicast group are usually derived from both a topology class and an arbitrary number of application classes. For instance, to model a flock of sheep a `Sheep` class describing an individual sheep would simply be combined by means of multiple inheritance with a topology class `Group` to a `FlockMember` class thus

describing the relations between an individual sheep and a (structured) group of sheep (fig. 3).

The members of the group do not need to be of the same exact type but need to be derived from the same member class. Thus full polymorphism is retained and each group member can potentially redefine all those multicast methods that have been declared as virtual methods.

3 Language-level Support

Figure 3 gives an impression how collective operations are supported on language-level. The `FlockMember` class inherits a `Sheep` class as well as the topology class `Group` to indicate that `FlockMember` is a group member class being able to process collective operations on groups of `Sheep`. Such classes can specify both group operations that are executed as collective operations on the whole group as well as (remote) methods that refer to the specific members only.

```
class FlockMember: public Sheep,
                   public Group</*!dual!*/FlockMember>
{
    public:
         // simple methods
         int rank() { return number; }

         // pattern (a) multicast without reduction
         void sayHello(String who) /*! multi !*/
             { cout << "Hello " << who << endl; }

         // pattern (b)(c) multicast with manual reduction
         void count(/*!dual!*/Counter result) /*! multi !*/
             { result.increment(); }

         // pattern (c) multicast with automatic reduction
         int countMembers() /*! multi:+ !*/
             { return 1; }
         ...
    private:
         int number;
}/*!dual!*/;
```

Fig. 3. A Member Class

Thus the `rank()` method of the `FlockMember` class refers to one member object only and returns a group related identifier of the object within the group. In contrast the `sayHello()`, `count()` and `countMembers()` methods are group related. While `sayHello()` causes all members to print a message to the screen, `count()` and `countMembers()` both calculate the number of objects in the group (fig. 3) in parallel.

Multicast methods can have arbitrary input parameters including proxy objects that allow the passing of global references to other objects. Proxy objects are identified by a `/*!dual!*/` annotation. Thus the `count()` method passes a `/*!dual!*/Counter` proxy to all group members who increment the counter object remotely. This is a simple example of a manual global reduction. In contrast the `countMembers()` method automatically reduces the multiple results in parallel. The syntactical notation `/*! multi:+ !*/` tells the stub generator that `countMembers()` is a multicast method and that its results should be reduced using the standard `+`-operator.

Multicasts are propagated in parallel according to the strategy defined by the topology class from which the member class is derived. The same holds for the reduction of the results. Thus the `countMembers()` method first is propagated in parallel to all members of the group according to the group's propagation strategy defined by the `Group` topology class. Then the multiple results are reduced in parallel according to the specified reduction method in the same order in which the multicast has been propagated. Provided this order is mathematically correct (refer also to section 4.2), any *binary* user defined reduction method that is compatible to the type of the result can be applied. The reduction method is applied iteratively when necessary. This makes the generated code independent from n-ary topologies producing `n > 2` subresults.

Instead of specifying the reduction method statically we could also have used a more dynamic scheme for reduction. Passing pointers to reduction methods to each multicast is even more flexible. This would have worked well for SPMD-programs but it is otherwise cumbersome to implement because the same method does not have the same address on each node. Furthermore, since also virtual methods of the group member class or any virtual method of the base classes can be specified for reduction, we think we found a scheme that is still very flexible and can effectively be implemented in any distributed or parallel system. This scheme can also be applied easily to implement parallel servers that are used by multiple clients.

3.1 Applying Object Groups

Whole object groups can be conveniently created through a `GroupOf` template class as depicted in figure 4. The declaration of `flock` as an instance of class

```
// create a flock of sheep with NumSheep members
GroupOf</*!dual!*/FlockMember> flock (NumSheep,"Sheep");

/*!dual!*/Counter counter;          // a counter Proxy

flock.sayHello("jon");              // pattern (a)
flock.count(counter);               // pattern (b)(c)
int sum = flock.countMembers();     // pattern (c)
```

Fig. 4. Applying Object Groups

`GroupOf</*!dual!*/FlockMember>` creates `NumSheep` instances of `FlockMember` at the parallel "Sheep" service and joins them to a distributed object group that now can be collectively addressed. Collective operations are issued to the group by simply invoking a method on the `flock` object representing the group. Thus the common client-server paradigm is extended to parallel services too. In principle group objects can also be passed as parameters to other (collective) methods thus granting access to existing object groups.

```
template<class T> class GroupOf: public T {
public:
   GroupOf(int nodes, const char* name):
       T(DomainAtNode(name, 0))
   {
       for (register int i = 1; i < nodes; i++) {
           join(new T (DomainAtNode(name,i)));
       }
   }
};
```

Fig. 5. The `GroupOf` Template

In figure 5 a simplified implementation for the `GroupOf` class is given. The template parameter `T` denotes the type of the group members. Each instance of T is created at the location denoted by `DomainAtNode()` that resolves the service identified by "name" at node `i`. Services are implemented by active server objects that are capable to handle object instantiations as well as invocations on existing objects. The code for the servers is automatically generated. Many clients can create objects at the same server and address these objects through proxy objects.

3.2 Topology Classes

If an application or distributed system service is already structured according to a specific topology, the existing communication infrastructure can easily be reused for global operations too. More specialized topology classes can be optimized for specific physical network topologies such as meshes or hierarchical crossbar networks. Tree topologies typically provide the best performance results in networks that offer roughly the same communication speed between any two nodes.

The methods of a topology class are used by automatically generated server stubs to control the order of the spreading of multicast messages and the number of results to expect. Topology classes (fig. 6) only need to define two methods used to control the spreading of multicast messages and the collection of the results.

The `members()`-method returns the number of group members or subgroup members in case of recursively defined topologies like lists or trees as shown in

```
class Topology  {
    public:
        virtual int members()=0;
        GlobalReference* iterate ()=0;
};
```

Fig. 6. Interface of a Topology Class

(fig. 7). The `iterate()`-method is used to determine the next group member to pass a multicast message to. This iterator iterates all group members and returns NIL when all members are iterated (fig. 7). Subsequent calls to the iterator start the iteration from scratch.

```
template<class T> class Group: public Topology {
public:
    void join(/*!in!*/ T* obj)
    {
        if (used == slots) {
            children[balance++%slots]->join(obj);
        } else {
            children[used++] = new T(*obj);
        }
    }

    int members() { return used; }

    GlobalReference* iterate()
    {
        T* result;
        if (iterated < members()) {
            result = children[iterated++];
        } else {
            result = NIL;
            iterated = 0;
        }
        return referenceOf(result);
    }
    ...
private:
    int used, iterated, slots, balance, ...;
    T** children;
}/*!dual!*/;
```

Fig. 7. A Group Class

Although only the `iterate()` and `members()` methods are needed by the automatically generated stubs, topology classes typically need to provide additional methods to add and remove members dynamically. The `GroupOf` class (fig. 5) e.g. requires all topology classes to provide at least a `join()` method as shown in figure 7 to build topologies dynamically. The `/*!in!*/` annotation of `obj` indicates that the argument `obj` needs to be passed by value. The `obj`

parameter is typically a pointer to a proxy to construct a distributed tree topology. When `obj` is locally inserted a copy of the proxy `obj` is made and locally stored. Otherwise `obj` is passed on to existing (remote) children applying `join` recursively.

Furthermore, user-level naming schemes that allow to address individual members of a multicast group by their group specific identifier instead of their system-level `GlobalReference` identifier often need to be supported. This is especially true for regular array topologies used for data parallel numerical computations that require indexing operations within n-dimensional index spaces. Such user-level naming can directly be supported, when the `GroupOf` class is slightly changed such that the remote references of the group members (the proxies) are stored in a table that maps user-level names to system-level references.

4 Multicast Processing

Multicast processing is entirely bound to the server's stub. Client stubs are not even aware that a specific method is executed as a multicast operation. In the most general case a server stub processes a multicast message as follows:

1. initialize a (local) barrier for the number of expected results.
2. allocate buffer space for the results.
3. spread the multicast message concurrently to other group members in the order defined by the `iterate()` method of the topology class
4. execute the desired method on the local object instance
5. wait for all results to arrive
6. reduce the results (including local result) according to the reduction method in the order previously defined by the `iterate()` method
7. reply reduced result to client

In case of simple multicast operations without results, steps 1-2 as well as 5-7 are omitted. The following subsections discuss the various handling schemes in detail and present several stub code examples.

4.1 Asynchronous and Triggered Multicasts

Simple multicasts without return values are handled in two ways. Either the multicast message is passed asynchronously to all (sub) group members or the multicast message is passed synchronously and each member sends an acknowledge before it starts to spread the message further and execute the desired method. We call the latter *triggered* multicasts and indicate such methods with a `/*! multi, trigger !*/` annotation. Triggered multicasts are non-buffered. When multiple casts are issued to a group a client only succeeds as far as the a previous cast has been completed. Thus multiple casts are pipelined into the group. Therefore a single source for multicast messages cannot overflow the network with numerous asynchronously sent messages.

```
void FlockMember::decode (ObjectRequest* msg ...)
{
    switch(msg->code) {
        ...
        case 1:
        {
            register FlockMember_sayHello_in1 *in;
            register GlobalReference *next;
            in = (FlockMember_sayHello_in1*) msg;
            while (next = iterate()) {
                next->pass(in->code, in, sizeof(*in));
            }
            this->sayHello(in->who);
        }
        break;
        ...
    }
}
```

Fig. 8. Server Stub for Asynchronous Multicasts

In contrast asynchronous multicasts are buffered and cause exceptions when the buffer space is exhausted. In figure 8 the server stub for an asynchronous multicast is depicted. The `decode()` method is executed when the basic ROI system propagates a message to an object. The stub generator generates a statically typed message format for each method and dependent on the method's numerical code the generic `ObjectRequest` message is casted to the specific format such as `FlockMember_sayHello_in1` in this example (fig. 8). Then the members of the group are iterated applying the `iterate()` method of the `Group` class. The message is propagated by a `pass()` method executed in the `while` loop. After propagation the requested multicast method is executed locally.

In case of triggered multicasts an acknowledge is sent to the invoker before the loop is executed and instead of the asynchronous `pass()` method a synchronous `invoke()` method is applied within the loop.

4.2 Global Reductions

Global reductions are implemented in two ways. Since proxy objects of other objects can be passed as arguments to multicasts each member can access these objects by reference and invoke methods on the remote object that will later hold the reduced result. This scheme can be extended with implicit lazy synchronization mechanisms if the passed object is a kind of future[6]. The benefit here is that the client can go on with computations while the multicast is processed. The disadvantage of this scheme is that the reduction of the result is serialized and can cost too much when either big multicast groups are involved (see section 5) or reduction methods are very complex.

Multicasts with parallel global reductions are more efficient in this case. Figure 9 shows a slightly simplified server stub for the `countMembers()` method with automatic parallel reduction. This is an example for a multicast that applies all

```
void FlockMember::decode (ObjectRequest* msg, DomainReference* invoker)
{ ...
        case 3:
        {
            register GlobalReference *next;
            register FlockMember_countMembers_in3 *in;
            register FlockMember_countMembers_out3 *out;
            FlockMember_countMembers_out3 reply_msg;
            in = (FlockMember_countMembers_in3*) msg;
            out = (FlockMember_countMembers_out3*) &reply_msg;

/* 1. */    Barrier results(members());
/* 2. */    FlockMember_countMembers_out3* replies =
                   new FlockMember_countMembers_out3 [members()];

/* 3. */    { for (register int i=0; next = iterate(); i++)
                  next->delegate(results, in->code,
                  in, sizeof(*in),
                  replies[i], sizeof(replies[i])); }

/* 4. */      out->result = this->countMembers();

/* 5. */      results.wait();
/* 6. */      { for (register int i=0;  i<members(); i++)
                  out->result = out->result+replies[i].result; }

/* 7. */    invoker->done(out, sizeof(*out));
            delete replies;
        }
        break;
... }
```

Fig. 9. Multicast with Reduction

steps described in section 4. After preparation of input and result messages a local barrier instance is initialized for the number of expected results (step 1 in fig. 9). Then buffers for all expected results from the (sub)group members are allocated (step 2). In the next step the message is casted to all (sub)group members by means of a `delegate()` method (step 3). This method sends a request and collects the result. When the result is available the passed barrier object is signaled. Our current implementation is straightforward and creates for each `delegate()` call a thread[1] that invokes the method synchronously. Thus multiple threads that invoke multiple methods concurrently are created implicitly in the spreading loop. After spreading the multicast message the `countMembers()` method is invoked on the local member (step 4.). Meanwhile the other group members handle the multicast concurrently.

When the computation of the local result is finished the flow of control is blocked by the `wait()` method until all results are available. We first wait for all results to arrive because we want to retain strictly the order in which results are processed. This is a strong requirement of numerical simulations. If we would compute the results in the order of arrival the result of a multicast would be

[1] Actually threads are managed efficiently in a pool and will be reused multiple times.

non-deterministic and two runs of the same program would not yield the same result in case of floating point operations executed in non-deterministic order. The order of reduction is the same as the order of message propagation and thus strictly defined by the user-controllable `iterate()` method of the topology class.

When all results are available the (partial) reduction of the results is computed in a `for` loop (step 6 in fig. 9) applying the specified reduction operation (the standard `+`-operator in this case). Finally the local result is replied (step 7) either to the client if the respective object was the group leader or to a parent group member that recursively continues the reduction until the group leader is reached.

5 Performance

All performance measurements have been performed on a 16 node Manna computer. The Manna is a scalable architecture based on a 50MHz i860 processor and is interconnected via a hierarchical crossbar network with 50MBytes/s links. Up to 16 nodes can communicate directly over a single crossbar switch. Therefore the communication latency between any two nodes is the same in the single cluster we used.

Our measurements where performed using the `FlockMember` class (fig. 3) introduced in section 3. The `Group` class topology is based on balanced n-ary trees. We have chosen a balanced binary tree for our measurements. This topology is not optimal as several researchers[4, 11] already have pointed out. Since we only want to show that our basic runtime mechanisms are scalable and efficient, we can neglect this fact. The objects of the group have been mapped to the computing nodes using a straightforward modulo 16 wrap around strategy.

We measured the five different types of multicasts that have been described in the previous sections:

1. an empty *triggered* multicast (`ping()`)
2. an empty *asynchronous* multicast (`pingAsync()`)
3. a *triggered* multicast with sequential reduction (`count()`)
4. an *asynchronous* multicast with sequential reduction (`countAsync()`)
5. a multicast with parallel global reduction (`countMembers()`)

All multicast messages fit into a single packet of 64 Bytes. Bigger messages are automatically transmitted using the bulk data transfer primitives supplied by the PEACE OS family. When messages do not fit into a single packet the startup-time for a single remote invocation is roughly doubled and additionally the transfer costs of the passed data need to be added on the basis of 40MBytes/sec (end-to-end on the Manna network).

Each multicast method has been issued 1000 times to achieve higher accuracy. Note that with two exceptions there have not been any pipeline effects caused by the fact that the same multicast was issued in a loop. The reason is that after each multicast with a result we always synchronized on the availability of the

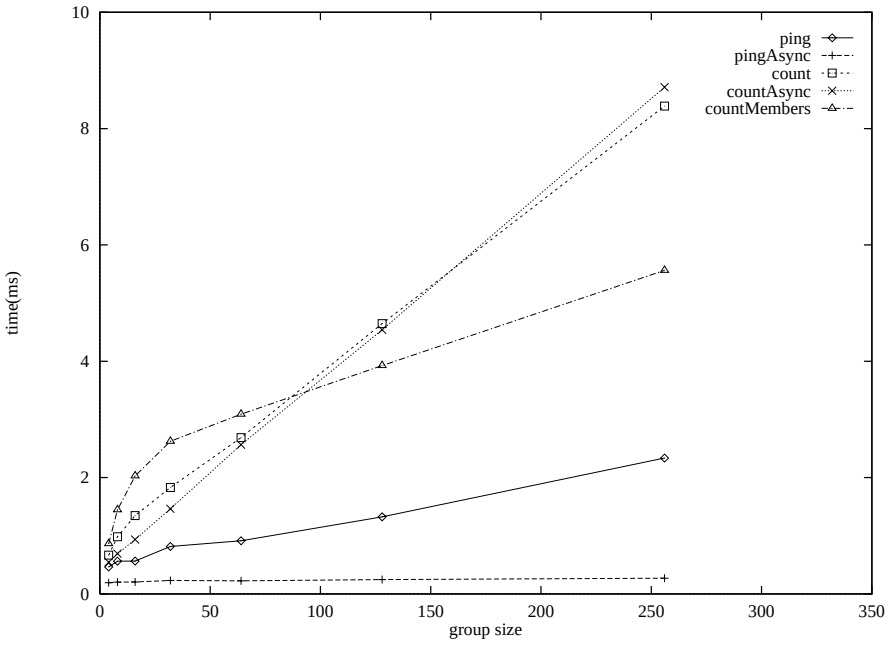

Fig. 10. Multicast performance for a balanced binary tree

result before we continued the loop. The asynchronous `pingAsync()` multicast possibly shows such effects, because we naturally could not synchronize exactly on the end of the asynchronous multicast. Therefore we issued only one multicast with one global reduction at the end to make sure, that all operations really had been performed. Consequently the measurement reveals some of the additional effects achievable with multiply overlapped multicasts issued to the same group of objects. The same holds for the triggered `ping()` multicast, which is implicitly flow-controlled but allows modest pipelining.

Figure 10 shows the results in detail. The `pingAsync()` method is extremely fast in the vicinity of 250 microseconds due to the pipelining reasons explained above. Notably the triggered `ping()` method that requires an additional acknowledge for each propagation step is still quite fast in relation to the amount of messages needed.

When looking at the methods that include reduction operations, the `countMembers()` method with parallel global reduction outperforms the other operations when object groups become larger. Initially the higher synchronization overhead for the parallel reduction is clearly the bottleneck for smaller groups. The `count()` and `countAsync()` methods with simple manual reduction via a proxy object passed as an argument in the multicast message only have minimal overhead. Therefore these methods are very efficient when applied to relatively small object groups. As expected, the sequential reduction of the

multiple results becomes the bottleneck when the object groups become larger because all group members send their local result to a single result object, that now has to process all messages. Here the implicit parallel global reduction is significantly faster and the graph shows a logarithmic shape as expected for a binary propagation tree. In contrast the curves for the methods with sequential reduction quickly convert to a linear shape revealing the bottleneck of the sequential reduction.

6 Conclusion

Multicast communication is neither new nor did we invent the notion of multicast topologies as such. Nevertheless we have shown that it is possible to implement flexible and efficient multicast communication applying relatively simple stub generator techniques in combination with the standard inheritance mechanisms of C++. The generator is designed to be used in conjunction with off the shelf C++ compilers and basically provides proxy based language-level support for computations in a global object space.

We found an easy way to provide effective control over group communication mechanisms through a simple iterator method that controls both the spreading of multicast messages as well as the reduction of the results. Therefore existing topologies can be reused for globally synchronized operations with modest effort. Furthermore, very specialized multicast patterns with minimal message exchanges can be implemented to solve e.g. problems like global page-invalidations in VSM systems.

Our collective operation patterns have much in common with those provided by the Ocore[8] programming language. Multicast groups have a similar functionality as *Communities* in Ocore or *Collections* in pC++[1]. The major difference is here, that e.g. communities have regular array topologies and message propagation cannot be controlled directly on application level using language-level constructs. Furthermore Ocore is based on a SPMD model like many other approaches targeting data-parallel programming such as pC++ or PROMOTER[10]. 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. The PROMOTER compiler extracts application patterns from topology definitions to optimize both alignment and mapping. 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.

We target more general parallel and distributed systems and like to support parallel servers that can be used by multiple applications. This scenario is specifically useful to increase multi-user performance when developers or scientists work on related topics and use parallel computers as domain-specific parallel servers for common parallel calculations. Therefore we do not initially support

complex data-parallel expressions based on index spaces. Nevertheless, topology classes allowing index based addressing can also be implemented with modest effort. Furthermore, many aspects of data-parallel programming are also covered by our approach as far as medium to coarse grained parallelism is concerned. Fine grained parallel computing like in ICC++[2] is possible in principle, but we do not expect satisfactory performance results since we don't apply aggressive compiler optimizations like in ICC++.

In the future we are considering to incorporate meta-object technology similar to MPC++[7] or Europa C++[14] in our design. Especially frameworks like Europa C++ could offer a reasonable basis for general purpose parallel and distributed systems since it does not rely on complex compiler technology but is still open to multiple computing paradigms.

References

1. Francois Bordin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), Fall 1993.
2. A. Chien, U.S. 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.
3. J. Cordsen. Basing Virtually Shared Memory on a Family of Consistency Models. In *Proceedings of the IPPS Workshop on Support for Large-Scale Shared Memory Architectures*, pages 58–72, Cancun, Mexico, April 26th, 1994.
4. 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.
5. Yvon Gourhant and Marc Shapiro. FOG/C++: a Fragmented–Object Generator. In *C++ Conference*, pages 63–74, San Francisco, CA (USA), April 1990. Usenix.
6. R. H. Jr. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985.
7. Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jörg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Reflection '96*, 1996.
8. 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.
9. Henry M. Levy and Ewan D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software—Practice and Experience*, 21(1):77–90, January 1991.
10. Besch M., Bi H., Enskonatus P., Heber G., and Wilhelmi M. 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.

11. J.-Y. L. Park, H.-A. Choi, N. Nupairoj, and L.M. Ni. Construction of Optimal Multicast Trees Based on the Parameterized Communication Model. Technical Report MSU-CPS-ACS-109, Michigan State University, 1996.
12. W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.
13. Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object–oriented operating system – assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.
14. The Europa WG. EUROPA Parallel C++ Specification. Technical report, http://www.dcs.kcl.ac.uk/EUROPA, 1997.