In the Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), pp. 1044 - 1053, June 30 - July 3, 1997, Las Vegas, USA.

Object-oriented Data Parallel Programming in C++

Hua Bi
RWCP Laboratory at GMD-FIRST
GMD Institute for Computer Architecture and Software Technology
Rudower Chaussee 5, 12489 Berlin, Germany
email: bi@first.gmd.de

Abstract *A lot of applications executed on distributed memory parallel computers can be classified as data parallel applications, in which parallel operations on data elements are performed in a loosely synchronous manner. However implementation of such applications is still difficult. This paper presents an object-oriented approach for an easy and efficient data parallel programming in C++. More exactly, an object model for distributed data is defined for modularity, polymorphism, and object sharing in programs, and data distribution and message passing details are encapsulated to reduce programming complexity and make reuse of code easier. Experiments show that this approach is efficient for a large class of data parallel applications.*

Keywords: data parallel programming, object orientation, SPMD paradigm, distributed memory parallel architecture

1 Introduction

By data parallel applications executed on distributed memory parallel computers, we mean applications in which a large set of data elements is partitioned on the distributed memory, and operations on those data elements are performed in the SPMD paradigm. The processes created by a SPMD program perform parallel computation locally; if the remote data elements are needed, they must be fetched before the computation.

However implementation of such applications is still difficult by using a message passing system like MPI or PVM. Object orientation is the best way to reduce programming complexity and write efficient programs. This paper introduces a programming approach to apply object orientation in implementation of data parallel applications. The underlying principles of this approach can be best described by the following features:

- An object model is introduced to specify distributed data by the principle of functionality decomposition. In this model, distributed data can be organized at three levels: topology, distribution and distributed object such that data distribution details are encapsulated. It leads to modularity, polymorphism and object sharing in programs
- Message passing details are encapsulated to reduce programming complexity and make reuse of code easier by decoupling communication patterns from communication itself.

This approach can be applied in a large class of applications, because the object model supports not only rectangular data topologies or shapes in applications, but also non-rectangular ones such as sparse or irregularly

[1]This work is supported by the Real World Computing Partnership (RWCP), Japan.

enumerated structures. By exploiting object orientation, this approach will lead to an easy data parallel programming, as well as an efficient implementation of data parallel applications.

In this paper we first present the object-oriented data parallel programming approach with a sample application in Section 2. In Section 3 our work is compared with other related work, and in Section 4 some performance results are given for conclusions.

2 Object-oriented Approach in SPMD Programming

Our approach assumes that a set of SPMD processes runs in parallel on a distributed memory parallel computer. Each process has its own control flow and its own address space. In our term, the address space of a process is called a domain. Each process can only access elements on its own domain, and can communicate with other processes by receiving and sending elements from and to other domains.

A domain can be identified by a domain number of the type Domain (typedef int Domain;) and a domain number ranges from *0* to *domain_max -1* . The global variable *domain_max* denotes the total number of domains, and the global variable *mydomain* denotes the domain number assigned to the own process.

2.1 Distributed Object

A distributed object is a collection of data elements which are partitioned and then mapped on the distributed memory. In many applications such as particle simulation, finite element method and computational fluid dynamics, problem-oriented data topologies or shapes may not only have rectangular spatial structures like arrays, but also non-rectangular ones. In order to specify an arbitrary spatial structure required by applications, one should be able to define an index space for a distributed object as an arbitrary finite subset of the power set Z^n, where Z is the set of all integers and n is an integer. Thus a Topology class is introduced to represent such an index space as a set of Points, where a Point is an n-dimensional integer vector:

```
typedef int* Point;

class Topology {
public:
    Topology( ... );
    ~Topology();
    int valid(const Point) const;
    static int dim() const; };
```

The method *valid* determines which point is a valid point in a topology. The dimension of a topology can be answered by the method *dim*.

To specify distributed data, in addition to a spatial structure we need to know how data elements are partitioned and mapped, and how data elements are stored and accessed at each domain. A Distribution class is introduced for this purpose:

```
class Distribution {
    const Topology& _topo;
public:
    Distribution(const Topology& tp, ...): _topo(tp) {...}
    ~Distribution();
    const Topology& topo() const { return _topo; }
    Domain domain(const Point) const;
    int index(const Point) const;
    int length(const Domain) const; };
```

A Distribution is a partitioned Topology. For a valid point p in a topology, the method *domain(p)* defines into which domain the point p is mapped. At each domain d, points are arranged in a sequence ranging from *0* to *length(d)-1* , and can be accessed by an index number *index(p)* for a point p. The pair *<domain, index>* is called point identification. Each point in a Topology can be uniquely expressed by the point identification which can be used in element allocation and element access.

Based on a distribution and an element type, distributed objects can be defined in the form of template classes as follows:

```
template<class DT, class ET> Disobj {
public:
    Disobj(const DT& t, const ET& init);
    ~Disobj();
    ET& operator[ ](const int index) const; };
```

That is, a distributed object can be defined with a distribution class DT and an element type ET, in the meaning that data elements are of the type ET, and data elements are partitioned and allocated according to the distribution class DT. The operator $[\,]$ accesses local data elements by an index number at *mydomain*.

A Disobj class can be implemented generically according to the information provided in a distribution and in an element type, that is, at each domain local data elements are allocated in a vector whose size can be calculated by the method *length* and the element size, and accessed by the method *index*. Because the frequently used operations on a distributed object are defined in a Distribution and they can be specifically implemented or optimized for a specific application on a specific machine, our implementation of distributed data will not impose a significant overhead.

A Topology encapsulates the spatial structure details, and a Distribution encapsulates the data partition and allocation details. Different objects of Distribution can share the same object of Topology, and different objects of Disobj can share the same object of Distribution. In this way, we provide an object-oriented representation of distributed data.

In the following, we give a simple example to show a triangular matrix as a topology and its block distribution. A Triangular class can be defined as follows:

```
class Triangular {
  int _n;
public:
  Triangular(const int n) {_n=n;}
  ~ Triangular() {}
  int valid(const Point p) const {
    if ((p[0]>=0) && (p[0]<_n) &&
        (p[1]>=0) && (p[1]<_n) &&
        (p[0]>=p[1])) return 1;
    else return 0;}
  static int dim() const { return 2;}
};
```

Figure 1 shows a triangular matrix Triangular(8) in which all points in this topology are given.

For a Triangular(n) , we can define many different distributions. The following code just gives a block distribution on the first dimension. Figure 2 shows the point identification for TriMat(8) , where $domain\_max = 4$.

<0, 0>
<1, 0> <1, 1>
<2, 0> <2, 1> <2, 2>
<3, 0> <3, 1> <3, 2> <3, 3>
<4, 0> <4, 1> <4, 2> <4, 3> <4, 4>
<5, 0> <5, 1> <5, 2> <5, 3>, <5, 4> <5, 5>
<6, 0> <6, 1> <6, 2> <6, 3> <6, 4> <6, 5> <6, 6>
<7, 0> <7, 1> <7, 2> <7, 3> <7, 4> <7, 5> <7, 6> <7, 7>

Figure 1: Topology: Triangular(8)

<0, 0>
<0, 1> <0, 2>
<1, 0> <1, 1> <1, 2>
<1, 3> <1, 4> <1, 5> <1, 6>
<2, 0> <2, 1> <2, 2> <2, 3> <2, 4>
<2, 5> <2, 6> <2, 7> <2, 8>, <2, 9> <2, a>
<3, 0> <3, 1> <3, 2> <3, 3> <3, 4> <3, 5> <3, 6>
<3, 7> <3, 8> <3, 9> <3, a> <3, b> <3, c> <3, d> <3, e>

Figure 2: Distribution: TriMat(8)

```
class TriMat {
  const Triangular& _topo;
  int blk_len;
public:
  TriMat(const Triangular& tp, const int n) : _topo(tp) {
    blk_len=n/domain_max; }
  ~TriMat() { }
  const Triangular& topo() const { return _topo; }
  Domain domain(const Point p) const {
    return p[0]/blk_len;}
  int index(const Point p) const {
    int _idx=p[1];
    int _row=domain(p)*blk_len;
    for(int i=p[0]-1; i>=_row; i--) {
      _idx + = i+1;}
    return _idx;}
  int length(const Domain d) const {
    return d*blk_len*blk_len+blk_len*(blk_len+1)/2; }
};
```

After defining a distribution, we can declare distributed objects which may share the same distribution as in the following code:

```
Tiangular tp(128);
```

```
TRiMat dis(tp, 128);
Disobj<TRiMat, double> x(dist, 0.0);
Disobj<TRiMat, int> y(dist, 0);
```

2.2 Parallel Computation

A data parallel application consists of a sequence of computational steps. At each step a collective communication can be performed at first, then a parallel computation can be executed completely locally, because the preceding communication has fetched the data needed to carry out the computation.

To simplify the problem, we assume that a parallel computation will be performed only on distributed objects with the same distribution, and element accesses on distributed objects in the computation should have the same alignment. If this is not the case, communication, which will be discussed in the next subsection, is needed.

With the above assumption, a parallel computation can be performed in an iteration over all selected local elements at each domain. A class of Local_Iter is introduced as an iterator on the index numbers of selected local elements as follows:

```
class Local_Iter {
public:
    Local_Iter(const Dist&, const Topo&...);
    ~Local_Iter();
    void rewind();
    int next(); };
```

Local_Iter(Dist, Topo) defines an iterator on the local index numbers with respect to a distribution *Dist* and a topology *Topo* which restricts the iteration on a subset of index numbers. The method *rewind* initializes the local iterator and moves to the first index at *mydomain*. The method *next* returns the current index number and moves to the next index at *mydomain*, or returns -1 if the iteration ends. The implementation of the above methods requires the data distribution information to transform data element addresses from a (global) point to a (local) index. Therefore Local_Iter encapsulates the localization details.

Because an object of Local_Iter is only dependent on a distribution, not on a distributed object, an object of Local_Iter can be reused by different operations on distributed objects, if these operations have the same execution pattern.

2.3 Communication

In our approach, a parallel computation is performed on distributed objects with the same distribution. In contrast a communication can be viewed as assignment between two distributed objects with different distributions. The implementation of such an assignment involves the data distribution and message passing details. As the first step of object orientation in communication, we introduce a Comm_Pattern class to encapsulate the localization details.

```
class Comm_Pattern {
public:
    Comm_Pattern(const Dist1&, Const Dist2&, ...);
    ~Comm_Pattern();
    Local_Iter send(const Domain);
    Local_Iter recv(const Domain); };
```

A Comm_Pattern takes one or two arguments of Distributions and possibly other arguments. The method *send(i)* returns a local iterator on all index numbers (relative to the one distribution) whose designated elements must be sent to the domain *i*, and the method *recv(i)* returns a local iterator on all index numbers (relative to the other distribution) whose designated elements will be operated with received data elements from the domain *i*, where i ranges from 0 to *domain_max-1*.

A communication pattern is defined as local observation on communication. In other words, the pattern defines the selection of data elements at *mydomain*. Such a pattern is problem-specific, and programming effort can be made to get an efficient implementation.

Because a communication pattern is only dependent on distributions, not on distributed objects, it provides the following three optimization possibilities in writing data parallel applications.

A communication pattern can be reused by different communication operations, if these

communication operations have the same communication relation and the same distribution relation among distributed objects. A communication pattern can be lifted out of a loop, if communication and computation in the loop do not change the spatial structure and data partition. The generation of communication patterns for the succeeding communications can be overlapped with the current communication, if these communication patterns are not dependent on the result of the current communication. In our experiments, the possibility to apply these optimizations is very high in many applications.

Based on communication patterns, the encapsulation of message passing details is provided by the following two template functions:

```
template< class DT1, class DT2, class CP,
          class ET1, class ET2, class FP>
void Expand(Disobj<DT1, ET1>&,
            const Disobj<DT2, ET2>&, CP&, FP&);

template<class DT1, class DT2, class CP,
         class ET1, class ET2, class FP>
void Reduce(Disobj<DT1, ET1>&,
            const Disobj<DT2, ET2>&, CP&, FP&);
```

In *Expand* are *Reduce*, elements of a distributed object of *<DT2, ET2>* should be sent to remote domains for the operation with the corresponding elements of a distributed object of *<DT1, ET1>*, according to a communication pattern of *CP* and an operation object of *FP* in which `void FP::operator()(ET1&, const ET2&)` is given. Both *Expand* are *Reduce* involve an all-to-all collective communication, and then a one-to-many or many-to-one operation respectively.

In message passing programming, programmers should consider the details about the creation of communication buffer and its management, the synchronization, and the data packing and unpacking. Such details are encapsulated within the functions *Expand* and *Reduce*, which can be implemented generically on top of a message passing system such as MPI or PVM. In the following, we describe one of their implementation frameworks called three-phase scheme.

In the first phase, all local data elements which must be sent to other remote domains are collected in a buffer and a non-blocking send is started.

In the second phase, the local operations are performed in a loop over index numbers defined by *send(mydomain)* and *recv(mydomain)*. The local operations are performed before receiving remote elements for the tolerance of communication delay.

In the third phase, it waits for any messages from all other domains. If a message from one domain is received, the relevant local data elements are operated with remote data elements according to an operation object. Because operations here are associative and commutative as defined in *Reduce* or single-target as defined in *Expand*, they can be performed just after the message from one domain arrives. In this way, computation is overlapped with the communication. In the third phase the synchronization is implicitly generated to wait for the completion of the overall collective communication. That is, it is a blocking receive for the collective communication, while messages from remote domains can be received in any order.

A distributed object may have a static element type like *int* or *double* or a dynamic element type (containing pointers) like a *tree*. Different code must be generated for packing and unpacking messages in *Expand* and *Reduce*. For distributed objects of dynamic element types, the class of these element types must have two methods *pack* and *unpack*. The method *pack* copies elements of a distributed object into a communication buffer, recursively copying all elements referenced through pointers. The method *unpack* restores an object from a communication buffer, recursively reassigning all pointers in the object with the correct linking value. For distributed objects with static element types, packing and unpacking are just simple memory copy.

2.4 A Sample Application

In this subsection we discuss how a sample application in particle simulation is programmed

by using this object-oriented approach. A Particle class is used to describe a particle in a two-dimensional space as follows:

```
class Particle {
public:
    double mass; // the mass of a particle
    double x_pos, y_pos; // the position of a particle
    double x_vel, y_vel; // the velocity of a particle
    Particle(...);
    ~Particle();
    int near(const Particle& another);
    void update(const Particle& another); };
```

The method *near* returns 1, if a particle is close enough to another particle such that they will collide; otherwise returns 0; If a particle collides with another particle, the method *update* updates its velocity and position.

A Particle_Bulk class is used to describe a collection of particles gathered in a subspace:

```
class Particle_Bulk {
public:
   int x_pos, y_pos; // the position of a particle bulk;
   Particle_Bulk(...);
   ~Particle_Bulk();
   Particle_Bulk& operator=(const Particle_Bulk&);
   int num() const; // the number of particles in the bulk
   particle& operator[](const int) const; // the ith particle
   void add(const Particle&); // add a particle to the bulk
   void null(); // clear the particle bulk
};
```

All particles at the position $< x, y >$ are gathered in a particle bulk at the position $< X, Y >$, if $\lfloor x/N \rfloor == X$ and $\lfloor y/N \rfloor == Y$, where N is an integer denoting the diameter of the subspace.

The following function *collide* describes that if paricles in a particle bulk collide with particles in another particle bulk, the velocity and position of all particles in this bulk will be updated:

```
void collide::operator()(Particle_Bulk& p,
                         const Particle_Bulk& q) {
    for(int i=0; i<p.num(); i++)
        for(int j=0; j<q.num(); j++)
            if (p[i].near(q[j])) p[i].collide(q[j]) }
```

After collision, some particles in a particle bulk will be moved to another particle bulk as described by the following function *move*:

```
void move::operator()(Particle_Bulk& p,
                      const Particle_Bulk& q) {
    for(int j=0; j<q.num(); j++)
        if ( (p.x_pos==(int)(q[j].x_pos/N)) &&
             (p.y_pos==(int)(q[j].y_pos/N)))  p.add(q[i]); }
```

Then the simulation of particle collision and movement can be described by the following code:

```
template <class Dist, class LIter, class CP >
void motion_phase(Disobj<Dist, Particle_Bulk>& x,
                  LIter& local, CP& cp) {
    Disobj<Dist, Particle_Bulk> y; // auxiliary object
    for(int i=0; i<Iter_No; i++) {
        copy(y, x, local); // copy x to y
        Reduce(y, x, cp, collide()); // update particles
        clear(x, local); // clear x
        Reduce(x, y, cp, move()); // move particles
    } }

template <class Dist, class LIter>
void copy(Disobj<Dist, Particle_Bulk>& y,
          const Disobj<Dist, Particle_Bulk>& x,
          LIter& local) {
    int j;
    local.rewind();
    while ((j=local.next())! =-1) {
        y[j].null();
        y[j]=x[j];} }

template <class Dist, class LIter >
void clear(Disobj<Dist, Particle_Bulk>& x,
           LIter& local) {
    int j;
    local.rewind();
    while ((j=local.next())! =-1) x[j].null(); }
```

The above code assumes that in a space of Particle_Bulks, particles collide with each other and move to its neighboring bulks (the neighborhood relation is described in *CP*).

The code is written without involving any distribution and message passing details. Message passing details are encapsulated in the function *Reduce*, and distribution details are encapsulated in the containing classes *Dist*, *LIter*, and *CP*. Because of encapsulation, the code is polymorphic for different *Dist*, *LIter* and *CP*, while *LIter* and *CP* are created once and used many times in a loop.

2.5 Communication Scheduling

The work to find a communication pattern for a communication is called communication scheduling. It is not easy to generate a communication pattern, because it involves the distribution details of distributed objects. In this subsection, we discuss generic implementation of communication scheduling by giving a logical communication relation as global observa-

tion which is more concise than a communication pattern as local observation.

A logical position of a data element in a distributed object can be described by a Point. Thus a logical communication relation can be defined as a mapping from a point to a list of points as follows:

```
template < int N, int M> class CommRel {
public:
    CommRel() { };
    ~CommRel() { };
    virtural PointList<M>& connect(const Point);
    static int indim() const {return N;}
    static int outdim() const { return M; }
};
```

CommRel<N, M> defines a logical communication relation form a N-dimensional Topology to a M-dimensional Topology. The method *connect* returns a list of M-dimensional points by giving a N-dimensional point p, in the meaning that the point p will be communicated with (assigned to, reduced from, or expanded to) all points in the point list returned. The class PointList is defined as follows:

```
template <int M> class PointList {
public:
  PointList(int sz);
  ~PointList();
  Point& operator[](int);
  int length() const ;
  static int dim() const { return M; }
  void rewind(int i=0);
  PointList<M>& operator<<(int);
  };
```

PointList<M> defines a list of M-dimensional points with the length of *sz*. A list of points can be written by the the operator << after applying the method *rewind(i)* to set the writing position to the *i-th* point, and can be read by the operator *[i]* for $0 \leq i <$ *length()*.

In the example of particle simulation, if the neighborhood relation in communication is defined in a two-dimensional topology as described in Figure 3, the logical communication relation then can be represented by the following class:

```
class Near : CommRel<2, 2> {
  PointList<2> _pl(8);
publlic:
```

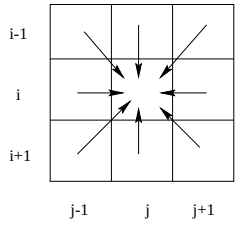

Figure 3: Neighborhood Relation

```
  PointList<2>& connect(const Point p) {
    _pl.rewind();
    return _pl << p[0] << p[1]+1
               << p[0] << p[1]-1
               << p[0]+1 << p[1]
               << p[0]-1 << p[1];
               << p[0]-1 << p[1]-1;
               << p[0]-1 << p[1]+1;
               << p[0]+1 << p[1]+1;
               << p[0]+1 << p[1]-1; }
};
```

Having a logical communication relation, the following template function can be used for communication scheduling:

```
template<class DT1, class RS1, class DT2, class RS2>
void CommSch(CP*, DT1&, RS1&, DT2&, RS2&, CR&);
```

CommSch generates a communication pattern of *CP* according to a logical communication relation of *CR*. The communication relation of *CR* defines a one-to-many communication relation from a distribution of *DT1* restricted by a topology of *RS1* to a distribution of *DT2* restricted by a topology of *RS2*.

CommSch can be implemented as a SPMD program, running in parallel at each domain as follows:

```
for each point p in Local_Iter(DT1, RS1) do
  for each point q in CR::connect(p) do
    if DT2::valid(q) and RS2::valid(q) then
      put q in a queue[DT2::domain(q)];
    endif
  endfor
  build DT1::index(p) into the iterator
  for CP::send(DT2::domain(q));
endfor

for each domain d except mydomain do
  send queue[d] to the domain d;
endfor

for each domain d except mydomain do
  receive queue[d] from the domain d;
```

```
  for each point q in received queue[d] do
    build DT2::index(q) into the iterator
    for CP::recv(d);
  endfor
endfor
```

In the above implementation, a local iterator *Local_Iter(DT1, RS1)* should be specifically implemented by involving the distribution details. The implementation of *CommSch* is generic with respect to the local iterator. The above implementation is also scalable, because the iteration number and communication amount decrease as the number of domains increases.

For an easy specification, a logical communication relation is always defined to be one-to-many. For a many-to-one communication from $< DT2, RS2 >$ to $< DT1, RS1 >$defined by *Expand*, a one-to-many communication relation from $< DT1, RS1 >$ to $< DT2, RS2 >$ can be directly passed to the *CommSch*. In this case, at *mydomain* the local iterators for *CP::send* are built directly, and the local iterator for *CP::recv(d)* is built according to a list of points received from the domain d, because these points are accumulated in consistency with a list of index numbers designating all elements to be sent from the domain d to *mydomain*.

For a one-to-many communication from $< DT2, RS2 >$ to $< DT1, RS1 >$ defined by *Reduce*, *DT1* and *DT2*, *RS1* and *RS2*, send and recv should be exchanged in the code of *CommSch*, because a one-to-many communication relation from $< DT2, RS2 >$ to $< DT1, RS1 >$ is given to *CommSch*.

In the example of particle simulation, a communication pattern for the communications in reduce/collide and reduce/move can be obtained with respect to a distribution *DIST* by the following code:

```
class sel_all {
public:
  int valid(const Point p) const { return 1;}
  static int dim()const { return 2;}
};

Near cr;
sel_all s;
Comm_Pattern c1;
DIST d1;

CommSch(&c1, d1, s, d1, s, cr);
```

As we see, a communication pattern is computed in a generic manner, that is, after defining a logical communication relation, it can be used for any kinds of distributions. The generic communication scheduling is less efficient than a direct implementation. Therefore only when communication scheduling is performed once and then reused many times in an application, it will not largely degrade the overall performance of an application. From our experience, a large class of data parallel applications such as in particle simulation, finite element method, and partial differential equation solver can be implemented in this way.

3 Related Work and Comparisons

Our approach is one of parallel C++ effort for data parallelism. It can be used at three levels: C++-level, library-level, and compiler-level. This paper just describes the approach used in C++-level without library support or compiler support. In the project PROMOTER[5, 1, 2], this approach is exploited at all three levels.

At a library-level, libraries for different Topology and Distribution classes can be provided, and runtime support to compute Local_Iter and Comm_Pattern with respect to defined Topology and Distribution classes can be used in writing data parallel programs. At a compiler-level, compile time techniques to construct Topology, Distribution, Local_Iter and Comm_Pattern can be developed. For example, a HPF compiler should generate Local_Iter and Comm_Pattern for a forall loop with distributed arrays with the (cycle) block distribution.

There are a lot of work in parallel C++ Efforts: ICC++ [4], C** [7], PC++ [8], CHAOS++ [3], HPC++[6], and others. Roughly speaking these efforts exploit two types of parallelism: data and task parallelism.

In our approach, only data parallelism is exploited and no task parallel efforts such as ob-

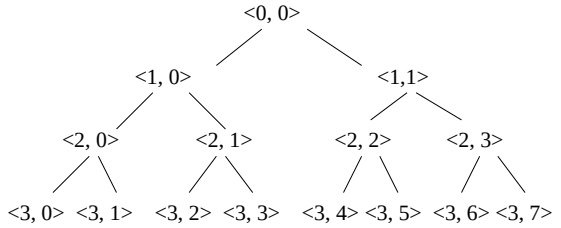

Figure 4: Topology: BinTree(4)

ject concurrency model have been made.

For data parallelism, collection, aggregate, or distributed array are used to represent parallel data structures in the other efforts. All of them can only specify rectangular structure. In our approach, arbitrary spatial structures are allowed. In other words, rectangular and non-rectangular (or irregular) spatial structures can be dealt with in our approach.

To support irregular data parallel applications, other approaches must use pointer-based data structures. For example, CHAOS++ provides support for distributed pointer-based data structures.

In our approach, such a pointer-based data structure is divided into two different worlds: topology and element type. In a topology, no pointers are allowed. This means that a structure like a tree must be flattened to a topology with all nodes in a tree. In an element type, pointers are allowed, but only possible to reference objects at the same domain. Therefore our data model is a restricted form of distributed pointer-based data structures, and we believe that many applications can be represented by this model, which can lead to a simple and efficient implementation for these applications.

To show how to flatten a pointer-based data structure, we define a topology BinTree for a binary tree (see Figure 4) as follows:

```
class BinTree {
  int _n;
public:
  BinTree(const int n) {_n=n;}
  ~ BinTree() {}
  int valid(const Point p) const {
    if ((p[0]>=0) && (p[0]<_n) &&
        (p[1]>=0) && (p[1]<pow(2,_n)))
        return 1;
    else return 0;}
  static int dim() const { return 2;}
};
```

This binary tree is embedded in a two-dimensional space. A data topology or shape can be embedded in different index spaces. In general, an index space selected for a data topology or shape must facilitate expression of logical communication relations. For example, a communication relation from a father to two sons in BinTree can be easily expressed by a one-to-many relation as follows:

$< i, j > \Rightarrow < i+1, j*2 >, < i+1, j*2+1 >.$

4 Conclusions

Our approach introduces a kind of object-oriented data parallel programming approach. At first we decouple distribution information from a distributed object itself such that a Distribution class can be implemented specifically for efficiency and a Disobj class can be implemented generically. Then we decouple a communication pattern from a communication such that a communication pattern can be implemented specifically also for efficiency, and a communication can be implemented generically. Lastly a communication pattern can be implemented generically under the condition that a corresponding local iterator is implemented specifically.

By using object orientation, a data parallel program can be written with modularity and polymorphism. Object orientation also leads to high performance, because it supports specialization, object sharing, and code reuse.

In our experiments, many applications such as heat conduction, fluid dynamics and finite element method have been or are being tested. The experiment results show that such applications can be efficiently implemented by using this approach, especially if a communication pattern generated can be used many times such as in a loop.

In the following we show the performance of three benchmark programs written by using

Table 1: Benchmark Performance

Bench-mark	4 Nodes PRO/PVM	8 Nodes PRO/PVM	16 Nodes PRO/PVM
MM	0.281/0.246	0.155/0.137	0.088/0.083
RL	0.280/0.261	0.137/0.132	0.075/0.072
CG	8.23/8.21	4.12/4.11	2.36/2.34

this approach and by directly using the message passing system PVM on the parallel machine MANNA[1]. Table 1 shows the times (seconds) needed for matrix multiplication (MM), relaxation (RL), and conjugate gradients (CG).

One of our future work is support of Dynamic Topology [9], that is, a distributed object can change its shape at runtime. It provides a conceptual equivalence to dynamic creation or expansion in pointer-based data structures. Dynamic topologies are mostly needed in the adaptive applications, in which a problem domain or a spatial structure has to be changed at runtime according to intermediate runtime results. In considering dynamic topologies we must also consider dynamic load balancing because distributions must also be dynamically changed according to changing topologies. Our preliminary work on these topics shows that efficient implementation of dynamic topologies seems to be possible, if some regularity in the adaption algorithm can be utilized.

References

[1] M. Besch, H. Bi, P. Enskonatus, G. Heber, M. Wilhelmi. High-level Data Parallel Programming in PROMOTER, In *Proc. of 2nd Int. Workshop on High-level Parallel Programming Models and Supportive Environments*, pp. 47-54, IEEE CS Press, April, 1997, Geneva, Switzerland.

[2] H. Bi. Towards Abstraction of Message Passing Programming . In *Proc. of Int. Conference on Advances on Parallel and Distributed Computing*, pp. 100-107, IEEE CS Press, March, 1997, Shanghai, China.

[3] C. Chang, J. Saltz, and A. Sussman. Chaos++: A runtime library for supporting distributed dynamic data structure. Technical Report CRPC-TR95624, Center for Res. on Parallel Computation, Rice University, Nov. 1995.

[4] A. A. Chien and J. Dolby. The illinois concert system: A problem-solving environment for irregular applications. In *Proc. of DAGS'94, The Sym. on Parallel Computation and Problem Solving Environments*, 1994.

[5] W. K. Giloi, M. Kessler, and A. Schramm. Promoter : A high level object-parallel programming language. In *Proc. of Internat. Conf. on High Performance Computing*, New Delhi, India, Dec. 1995.

[6] G. HPC. Hpc++ white paper. Technical Report CRPC-TR95633, Center for Res. on Parallel Computation, Rice University, 1995.

[7] L. R. Larus. A large-grain, object-oriented data-parallel programming language. In U. Banerjee, A. N. D. Gelernter, and D. Padua, editors, *Languages and Compilers for Parallel Computing (5th International Workshop)*, pages 326–341. Springer-Verlag, Aug. 1993.

[8] A. Malony, B. Mohr, D. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan. A parallel c++ runtime system for scalable parallel systems. In *Proc. of Supercomputing'93*, pages 140–152. IEEE CS. Press, Nov. 1993.

[9] A. Schramm. Irregular applications in promoter. In W. K. Giloi, S. Jaenichen, and B. Shriver, editors, *Proc. of Internat. MPPM Conference*, Berlin, Germany, Oct. 1995. IEEE CS. Press.

[1] MANNA is a scalable parallel machine with distributed memory. See http://www.first.gmd.de.