Distributing Object Arrays in C++[1]

Lubomir Bic

Department of Information and Computer Science
University of California, Irvine, CA 92717

bic@ics.uci.edu, (714) 856-5248

Technical Report 90-35

October 18, 1990

Abstract

Object-oriented programming, due to its encapsulation and message-passing principles, is a suitable basis for the programming of multicomputers. We present mechanisms which allow sequential C++ programs to be transformed into parallel programs, executable on an MIMD architecture, such as the Intel iPSC/2 Hypercube. The basic idea is to provide mechanical transformations, which allow arrays of objects to be distributed over different processors. At run time, a master processor may command other processors, considered slaves, to each create a different subrange of the original array. Subsequently, it may command the slave processors to execute any member functions of the remote array elements. Parallelism is achieved by segregating the send and receive commands issued by the master, which allows the slave processors to operate concurrently.

Keywords: object-priented programming, parallel processing, C++, MIMD architectures

[1] This work was supported by the NSF Grant CCR-8709817

Contents

1. Introduction

There are two basic principles that make object-oriented programming attractive for parallel computer systems. The first is *encapsulation*, which packages data and procedures into objects, thus implementing abstract data types [GUT77]. Hence, the problems of data-distribution and program-distribution are merged into one. Furthermore, there are no side-effects in purely object-oriented programs, since all computations are confined within objects. This greatly simplifies the data-dependency analysis necessary to determine possible distribution.

The second principle is *message-driven* computation, which is used (at least conceptually) in sequential object-oriented programs. That is, member functions are invoked by "receiving a mesage". This point of view translates naturally into a distributed environment, where messages are not only conceptual, but actually travel between different processors.

Due to the above principles, a number of projects have been started to implement parallel object-oriented languages and systems [AME87, BLL88, BOLA89, DACH88, GERO88, POLE89, YOTO87]. The main thrust of these projects is to extend existing object-oriented languages by providing constructs for parallelism, or to create new parallel object-oriented languages, again with explicit control over parallelism. Our philosophy, on the other hand, is to develop tools to perform automatically as much of the parallelization of a program as possible. In [YBU90A, YBU90B], we have proposed transformations for automatically distributing C++ programs consisting of only static objects, i.e., objects known at compile time. The transformations presented in this paper are a first step toward distributing *dynamic* objects, i.e., objects defined using the construct *new*, or objects defined automatically when their scope is entered.

We concentrate on distributing *arrays* of objects, rather than individually defined ones, as most parallelism typically results from iterating over arrays. The proposed transformations, presented in Section 2, allow a processor to command other (slave) processors to create subranges of a given array and to invoke their member functions remotely. By segregating the send and receive commands, concurrency among the slave processors is achieved, as will be discussed in Section 3. An application example to illustrate the capabilities of the proposed transformations is then given in Section 4, followed by conclusions in Section 5.

2. Transformations to Achieve Distribution

This section considers the mechanisms for achieving distribution. For any array to be distributed, these mechanisms automatically transform the declaration of that array and all references to it such that the array elements are distributed over all PEs and the accesses work properly using send and receive primitives.

2.1. Basic Assumptions

The distribution works in a master/slave relationship. For simplicity, the following assumptions are made:

- The declaration of the array (A) and all accesses to it will be executed on one processing element, say PE0. This becomes the master PE.
- The array (A) will be distributed equally over the remaining $np-1$ PEs, i.e., PE_1 through PE_{np-1}. These become the slave PEs. Assuming the size of the array is a multiple of $np-1$, each PE will hold a subrange $\text{A}[k, 2k-1]$, where k is a multiple of $n/(np-1)$.
- The main() function of all slave PEs consists of an infinite loop, which repeatedly receives a message and, using a switch statement, performs the requested action. The general structure is:

```
class c
{...
  type data1;                    //general form of a data member
  res_type method1(param){...}  //general form of a method
 ...
}

...
c A[n];     //declaration of array of objects of type c somewhere
            //in the program
...
res1 = A[i].method1(param);  //method invocation of some A[i]
                             //somewhere in the program
res2 = ... A[j].data1 ...;   //read access to data member of A[j]
                             //somewhere in the program
A[k].data1 = expression;     //write access to data member of A[k]
                             //somewhere in the program
```

Figure 1

Sequential program

```
while (1){
  rcv msg;
  switch (command)  //each message carries a string as its first
                    //component, which determines what to do
  ...
  case "xxx": perform action corresponding to xxx
  ...
}
```

- The general program structure before distribution is as shown in Figure 1. It contains a class definition, a declaration of an array of objects of that class, a method invocation, and a read and a write access to a data member. In the method invocation we assume that *param* is a set of parameters, all of which are passed by value. Call by reference (pointer) requires additional mechanisms (e.g., sending a copy of the data to the callee, locking it while it is being accessed by the callee, receiving the modified version, updating and unlocking the the original version). This issue is not dealt with in this paper.

2.2. Identifying Remote Objects

In this paper we are concerned with arrays of objects. Conceptually, each array can be viewed as a complex object whose components are individual element objects. In a sequential C++ program, each array is uniquely referred to by its name or by a pointer. Individual elements are referred to by their index, which in a shared memory environment is used to compute the location of the desired element.

In a distributed environment, this is no longer possible, since different array elements reside in different address spaces. This means that all references to remote elements must be translated into send/receive primitives, and the messages must carry information to uniquely establish the relationship between the elements belonging to the same array. This must be done dynamically, since objects are created and destroyed automatically as code blocks are entered and exited, or explicitly, using the "new" and "delete" statements.

To provide for unique identification of distributed components we take advantage of the pointer "this", which is maintained implicitly by C++ for all objects [STR87]. It corresponds to the address of the object in memory and hence is unique as long as the object is active. We use this pointer to establish the correspondence between the elements constituting the same array. It is interpreted as a long integer (i.e., not dereferenced) and is carried on messages between the master and the slave PEs when creating or accessing the remote subranges of a distributed array.

The implementation is as shown in Figure 2. Each slave PE maintains an array of pointers, ptr[ ]. Each entry corresponds to a subrange of a distributed array. The assignments of the different subranges to entries in ptr[ ] are done dynamically, using the pointer value "this". However, since the size of "this" is very large (size of memory address), hashing is used to obtain an index within the bounds of ptr[ ].

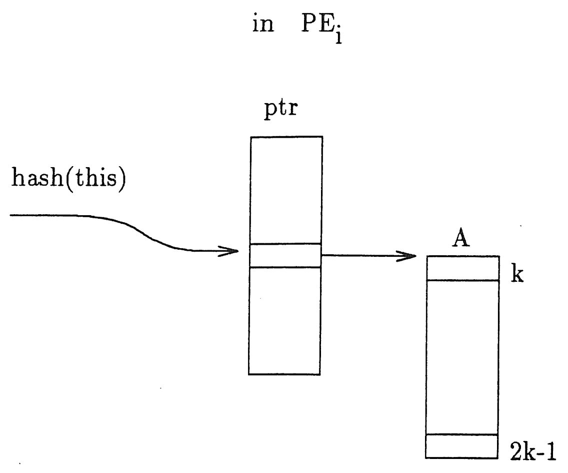

Figure 2

Identifying remote subarrays

Hence distributed arrays are accessed by each slave PE using ptr[hash(this)], where "this" is carried on each message from the master PE.

Note that the actual implementation is somewhat more complicated due to possible collisions. That is, the ptr[ ] array must either handle lists of objects whose "this" pointer hashes to the same index, or some other overflow mechanism must be provided. To simplify the subsequent presentation, the notation ptr[hash(this)] will be used to denote the unique pointer to a given array subrange.

2.3. The Transformations

The transformation of a sequential C++ program consists of several steps, as described next:

(1) Define New Class for c

For each class c to be distributed, provide a new class c_d. The suffix "d" indicates distribution. The original class c is preserved to also allow the creation of centralized instances of c, if desired.

The new class c_d has the following general structure:

```
class c_d;
{...
  c_d(n)        //constructor
  { for (i = 1; i < np; ++i)
      send to PEi: <"create c", this, n>; }
}
```

The new class essentially consists of a constructor, which takes n— the number of elements in the original array declaration—as a parameter. It then sends a message to each slave PE, which carries the command "create c" (where c is the original class name), the pointer "this", and the number n. The command will determine the action to be taken by the slave PE. (Note that it is not necessary to carry the actual string "create c", but only some internal encoding of it. Here we use the string for clarity.) The pointer "this" is used to identify the corresponding subranges in the slave PEs. As explained in Section 2.2, its content, which is an address in the master PE's memory, is sent to all slave PE, where it is interpreted as a long integer and its hash value is used to index the array ptr[ ].

(2) Extend Switch Statement in Slave PEs

In each slave PE, the switch statement is extended by the following new case:

```
while (1){
  rcv msg;
  switch command;
  ...
  case "create c": ptr[hash(this)] = new c[n/(np-1)];
  ...
}
```

This causes the creation of a new array of objects of type c but the range of this new array is a fraction of the original range, $n/(np-1)$, where n is the original range and np is the number of PEs. The pointer to this new array is stored in an array ptr[ ], at the index derived from "this".

(3) Transform Array Declarations

Each declaration of the original array, i.e., c A[n];

which is to be distributed, is replaced with

c_d A(n);

Note the different parenthesis around n. This new declaration now creates a *single* object, A, of *type c_d*, and passes n to its constructor. This then causes the creation of the *subarrays* of *type c* in the slave PEs, as discussed above.

(4) Provide New Methods for Data Members of Class c

For each data member d in the original class c provide two new methods, "put_d" and "get_d" in c. The new methods have the form:

```
res_type get_d()
{ return d; }

void put_d(val)
{ d = val; }
```

This allows us to refer to all members uniformly using a functional notation, m(), regardless of whether m is a method or data. That is, read references to data members may be replaced by the call get_data(), while write references are replaced by put_d(). (The need for this extension will become apparent in Step 8 below.)

(5) Provide Methods for Class c_d

For each method m of the original class c (including the new methods get_d and put_d) provide two corresponding method m_s and m_r in c_d. The extensions s and r stand for "send" and "receive". The reason for creating two separate methods is to achieve concurrency during execution, as will be explained in Section 3.

Assuming that the original method is of the form:

```
res_type m(param) {...}
```

then the new methods have the form

```
void m_s(i, param)
{ send to PE holding i: <"m", i, this, param> }
```

```
res_type  m_r(i) {
  receive res;
  return res
}
```

The new parameter i results from replacing references to the original array elements A[i] by references to the single (distributed) object A (see Step 7).

Note that either param or res could be void. If param is void, the function m_s has the form:

```
void m_s(i)
{ send to PE holding i: <"m", i, this> }
```

If res is void, the function m_r has the form:

```
void m_r(i)
  { receive ack; }
```

where ack represents an acknowledgement. This may or may not be necessary to preserve the original semantics of the program. Depending on the data dependencies in the program, subsequent calls could interfere with m's operation if no acknowledgement is requested.

(6) Extend Switch Statement in Slave PEs

The switch statement in all slave PE is extended as follows. For each method m of c, including get_d and put_d methods, provide a new case statement of the form:

```
...
  switch command;
  ...
  case "m":
     res = ptr[hash(this)][i % (np-1)].m(param);
     send res to master;
  ...
}
```

The command "m" specifies that the method m is to be invoked. The array subrange is located by using the hash value of the received "this" pointer as index into ptr[ ]. The array element for which m is invoked is given by i modulo the number of slave PEs. This is because each subrange created by a slave PE starts with the index 0 (this is a general limitation of C/C++). The original parameters, if any, are passed to m unchanged and the result (or acknowledgment) is returned to the master PE.

(7) Transform References to Methods

References to methods cannot occur on the left-hand side of assignment statements. (This is due to the assumption that return results are not references but only values). Hence all references to methods have the form

```
res = A[i].m(param);
```

where res could be an implicit variable (invocation withing an expression). Each such reference is replaced with the following two calls:

```
A.m_s(i, param);
res = A.m_r(i);
```

The first statement invokes the method m_s in the new object A, which sends the request to the corresponding PE holding A[i] to execute the original method m on that element. The second statement then invokes the method m_r of A, which receives and returns the result.

(8) Transform References to Data Members

References to data members may occur on either side of an assignment statement. They are transformed as follows:

Each read reference to d, i.e.,

```
... = ... A[i].d ...
```

is replaced by

```
A.get_d_s(i);
... = ... A.get_d_r() ...
```

Similarly, each write reference to d, i.e.,

```
A[i].d = exp
```

is replaced by

```
A.put_d_s(i, exp);
ack = A.put_d_r();
```

Figure 3 shows the complete program resulting from applying the above transformations to the original program of Figure 1. The slave PEs execute the code shown in Figure 4.

Note that the transformations as presented so far implement a form of Remote Procedure Calls, where the master invokes the methods of other objects, located on other PEs. A crucial point of this implementation is that the call is split into two separate procedures – one to invoke the remote method and to send it the necessary parameters, and the second to receive the results. This split-phase implementation is the basis for achieving concurrency, as will be discussed in Section 3.

2.4. Nested Objects

The above mechanisms for distributing arrays of objects work also for arrays nested within other objects. The transformations shown in Figue 5 would be applied if the array were nested within another class, called "outer".

3. Concurrency

The main goal of distributing objects is to achieve concurrency and thus reduce overall computation time. The transformations as presented so far, achieve this goal only to a very limited extent. They permit us to distribute objects. However, since all method invocations (and data member references) are translated into m_s() immediately followed by m_r(), only one slave PE could be active at any one time.

```
class c
{...
  type1 data1;                     //general form of a data member
  res_type1 method1(param){...}  //general form of a method
 ...
  res_type2 get_data1()  //new get method for data1
  { return data1; }
  void put_data1(val)        //new put method for data1
  { data1 = val; }
}

class c_d;
{...
  c_d(n)        //constructor
  { for (i = 1; i < np; ++i)
      send to PEi: <"create c", this, n>; }

   ...
  void method1_s(i, param)  //new send method for original method1
   { send to PE holding i: <"method1", i, this, param> }

  res_type1 method1_r(i) {   //new receive method for original method1
     receive res;
     return res
   }

  void get_data1_s(i)  //new send method for get_data1
   { send to PE holding i: <"get_data1", i, this> }

  res_type2 get_data1_r(i) {   //new receive method for get_data1
     receive res;
     return res
   }

  void put_data1_s(i, param)  //new send method for put_data1
   { send to PE holding i: <"put_data1", i, this, param> }

  void put_data1_r(i) {   //new receive method for put_data1
     receive ack;
   }
}

...

c_d A(n);  //new declaration

res1 = A.method1(i, param);      //new method invocation

res2 = ... A.get_data1(j) ...;   //new read reference to A[j]

A.put_data1(k, expression);      //new write reference to A[k]
```

Figure 3

Transformed program

```
while (1){
  rcv msg;
  switch command;
  ...
  case "create c": ptr[hash(this)] = new c[n/(np-1)];
  ...
  case "method1":
     res = ptr[hash(this)][i % (np-1)].method1(param);
     send res to master;

  case "get_data1":
     res = ptr[hash(this)][i % (np-1)].get_data1();
     send res to master;

  case "put_data1":
     ptr[hash(this)][i % (np-1)].put_data1(param);
     send ack to master;
  ...
}
```

Figure 4

Code for slave PEs

```
   before                 after

class outer            class outer
{ c A[n];              { c_d A(n);
   ...                     ...
}                      }

outer O;               outer O;    //declaration

A[i].m(...);           A.m(i, ...);    //reference within O

O.A[i].m(...);         O.A.m(i, ...);    //reference outside of O
```

Figure 5

Transformation for nested objects

This is because m_r() contains a blocking receive statement. Hence a sequence of references to different objects, e.g. a loop of the form

```
for (i = 1; i < np; i++) {
  A[i].m(...);
```

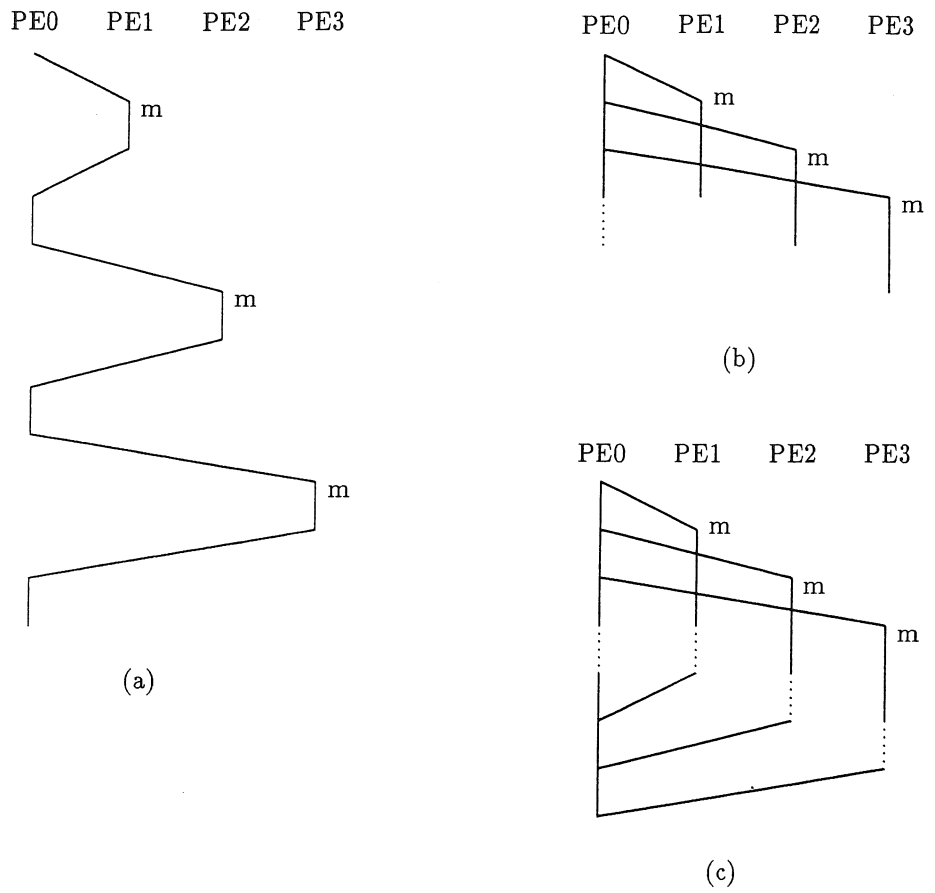

Figure 6

Possible concurrency

}

would result in a sequential execution (with additional communication overhead) as shown graphically in Figure 6(a).

In cases where no return result is expected, we could eliminate the call to m_r(), which would allow the next m_s() to proceed immediately. The resulting behavior is depicted in Figure 6(b). This, however, does not result in a very useful environment, since operations that do not return any results are limited essentially to initializations.

To achieve concurrency in general, it is necessary to segregate the send and receive statements, so that all slaves can first be commanded to start some computation and only then the master blocks to await the results. The transformation described in Section 2.3 have been designed with this objective in mind. In particular, each reference to a remote method or data member is transformed into two calls – one that performs the send and the other to perform the receive. This allows us to perform an additional transformation on the program which essentially changes the order of invocation of the two methods.

With straight-line code, all references occur explicitly and may simply be reordered to segregate the send and receive calls. For example, the sequence

```
r1 = A[1].m(...);
r2 = A[2].m(...);
r3 = A[3].m(...);
...
```

is first transformed into the following calls:

```
A.m_s(1, ...);
r1 = A.m_r(1);
A.m_s(2, ...);
r2 = A.m_r(2);
A.m_s(3, ...);
r3 = A.m_r(3);
...
```

The segregation transformation then yields:

```
A.m_s(1, ...);
A.m_s(2, ...);
A.m_s(3, ...);
r1 = A.m_r(1);
r2 = A.m_r(2);
r3 = A.m_r(3);
...
```

which results in the execution profile shown in Figure 6(c). The result is a pipeline which overlaps the execution of the different slave PE.

With loops, the transformation is more complicated. The original loop must be divided into two loops, one performing the sends and the other the receives. Consider for example the following loop:

```
for (i = ...)
  { res[i] = A[i].m(...); }
```

The original transformation yields:

```
for (i = ...) {
  A.m_s(i, ...);
  res[i] = A.m_r(i); }
```

The segregation transformation yields:

```
for (i = ...)
  { A.m_s(i, ...); }

for (i = ...)
  { res[i] = A.m_r(i); }
```

which has the desired behavior shown in Figure 6(c).

With the additional transformation, concurrency is obviously achievable. The main problem, however, is to guarantee that the resulting program still conforms to the original semantics of the sequential program. If that is not the case, the transformation cannot be performed.

In general, this requires an interprocedural analysis to be carried out to determine whether the execution of one method depends in any way on the execution of another within the sequence to be transformed. Due to encapsulation in purely object-oriented programs, this analysis is considerably simpler than with conventional programming languages. In particular, in the absence of any global data it is much easier to determine the read and write sets of each object and to determine whether there is any possible interaction between them. In addition, there are a number of spacial cases, which are verified relatively easily. Below, we consider two such cases that occur frequently in C++.

Case 1:

Two consecutive statements of the form:

```
A1.m1(y1);
A2.m2(y2);
```

are independent and thus may be interchanged if the following conditions hold:

1. A1 and A2 are distinct objects. This is true when A1 and A2 are different explicit names (i.e., not pointers), or when A1 and A2 are array elements and A1 and A2 are distinct names or their respective indices have different values.
2. A1.m1 or A2.m2 does not reference any object global to A1 or A2. Referencing a global object can only be done by explicitly naming that object, and thus is easily checked.
3. All parameters are passed by value (including those passed to A1 and A2 at the time of creation). This is easily checked by examining the source code.

Condition 1 guarantees that any local data accessed by m1 and m2 belongs to physically disjoint sets. Condition 2 guarantees that m1 and m2 do not access any common object. Condition 3 guarantees that any data passed to m1 and m2 as parameters at the time of invocation, or to A1 and A2 at the time of creation, cannot be shared.

Together the three conditions guarantee that there is no possible interaction among the two statements and hence their execution can be interleaved by first invoking the sending methods (A.m_s()) followed by the receiving methods (A.m_r()).

Case 2:

Two consecutive statements of the form:

```
x1 = A1.m1(y1);
x2 = A2.m2(y2);
```

are independent and thus may be interchanged if, in addition to the same three conditions of Case 1 above, the following fourth condition holds:

4. There is no data dependence between the parameters x1, x2, y1, and y2. This is true if the parameters are simple variables v_i or elements of explicitly declared arrays, $r_i[k_j]$, and the following holds: for any pair of simple variables the names v_i are distinct; for any pair of array elements the names r_i are distinct or the indices k_j have different values.

   This condition guarantees that the reading of the parameters y1 and y2 does not depend on the writing of the output values x1 and x2.

Similar to Case 1, the four conditions guarantee that there is no possible interaction among the two statements and hence their execution can be interleaved.

4. An Application Example

The Compute-Aggregate-Broadcast (CAB) paradigm [NESN87] is a well know structuring techniques for solving problems on multicomputers. The basic idea is to partition a problem into subproblems to be partially solved on individual PEs of the multicomputer. During the compute phase, each PE works on its own subproblem, producing some partial result. In the aggregate phase, these results are combined into a result for the overall problem. If this results satisfies some given criterion, the computation stops. Otherwise, the broadcast phase is entered, which distributes the data necessary for the next compute phase to individual PEs. The compute-aggregate-broadcast cycle is repeated until the desired result is obtained. The CAB paradigm generally operates in a master-slave fashion, where the master evaluates the termination condition and commands the slave to continue, as appropriate.

There are different forms of the CAB paradigm, depending on how the data is distributed and exchanged among individual PE. In its simplest form, data is

```
class c
{  local variables;

   void initialize(p)     //method
   { store p in local variables; }

   res_type compute(p)    //method
   { compute partial result using p and local data;
     return result;
   };

main()
{ ...
  c A[n];
  for (i = 0; i < n; i++)
    { A[i].initialize(p); }
  while (result not satisfactory) {
    for (i = 0; i < n; i++) {
      res[i] = A[i].compute(p);
    }
    p = init;
    for (i = 0; i < n; i++) {
      p = aggregate(p, res[i]);
  }
}
```

Figure 7

Sequential CAB program

exchanged directly only between the master and the slave PEs. That is, the master aggregates all partial results produced by slave PEs, evaluates the termination condition, and, if another iteration is to be performed, it broadcasts the aggregated data to all slave PE. Hence slave PEs only perform the compute phase.

The program skeleton in Figure 7 is a sequential C++ program that mimics the behavior of a CAB program. For simplicity, we assume that $n = np-1$, i.e., the size of the array is equal to the number of slave PEs. Hence, after transformation, each slave will hold exactly one object A[i].

The main() function plays the role of the master. After initializing all A[i] elements by sending them the data p, it invokes the compute() function of each object A[i]. This corresponds to the broadcast phase, which distributes a copy of the parameter p to each object. The results produced by the individual objects are collected in the array res[ ] and are subsequently combined into a new value for p using the function aggregate(). This is then broadcast to all slaves during the next iteration of the while loop.

The slave objects, A[i], are instances of the class c, which contains the methods initialize() and compute(). The latter produces the partial solution using the parameter p and its local data. This solution is returned to the master.

In the following, we demonstrate that by applying the transformations presented in Section 2.3 to the above "CAB" program, a computation that actually displays the desirable characteristics of the parallel programming paradigm is automatically derived.

Figure 8 shows the new program after transformation. It comprises the new class c_d with the corresponding methods initialize_s(), initialize_r(), compute_s, and compute_r, derived by following the algorithm of Section 2.3. In the main() function, the declaration of the array has been changed to c_d A(n) and the references to A[i].initialize(...) and A[i].compute(...) have been replaced by the pairs of calls A.initialize_s(i, ...), A.initialize_r(i, ...), and A.compute_s(i, ...), A.compute_r(i, ...), respectively. The send calls and receive calls have been segregated into separate loops to achieve parallelism. The segregation of the initialization calls is legal according to Special Case 1, described in Section 3. Any two successive calls

```
A[i].initialize(p);
A[i+1].initialize(p);
```

satisfy the following conditions and thus do not depend on each other:

1. Each calls a different object
2. initialize() does not reference any global object

```
class c { ... }  //contains new methods of the form get_d and put_d
                 //for all its data members

class c_d
{
    c_d(n)  //constructor
    { for (i = 1; i < np-1; i++) {
        send to PE holding i: <"create c", this, n>
       }
    }

    void initialize_s(p)
    { send to PE holding i: <"initialize", i, this, p> }

    void initialize_r()
    { receive ack; }

    void compute_s(p)
    { send to PE holding i: <"compute", i, this, p> }

    res compute_r()
    {  receive res;
       return res
    }
    ...
}

main()
{ ...
  c_d A(n);

  for (i = 0; i < n; i++) {       //transformed initialization loop
      A.initialize_s(i, p);
  }
  for (i = 0; i < n; i++) {
      A.initialize_r(i);
  }
  while (result not satisfactory) {
    for (i = 0; i < n; i++) {    //transformed computation loop
      A.compute_s(i, p);
    }
    for (i = 0; i < n; i++) {
      res[i] = A.compute_r(i);
    }
    p = init;
    for (i = 0; i < n; i++) {
      p = aggregate(p, res[i]);
  }
}
```

Figure 8

Parallel CAB program

3. p is passed by value and no parameters were passed to A at the time of creation

Similarly, the segregation of the computation calls is legal according to Special Case 2. Any two successive calls

```
    res[i] = A[i].compute(p);
    res[i+1] = A[i+1].compute(p);
```

satisfy the first three conditions in the same way as the initialization calls. The fourth condition is also satisfied, assuming that p is a simple variable or an element of an array different from r. Then there is no data dependence between p and res[i] for any i.

5. Conclusions

With the mechanisms described in this paper it is possible to create and operate on distributed arrays in C++. The transformation from a centralized to a distrubuted array is fully automatic. At run time, the master PE commands the slave PEs to define subranges of the original array and to apply the methods stated in the original program to their respective subranges.

Concurrency is achieved if the sends to the slaves may be segregated from the receives. For example, a loop that invokes A[i].m(...) for a different i in each iteration could be divided into two loops; one performing the sends and the next performing the receives, thus achieving a pipelining effect. This is possible if there is no interaction between the different elements A[i]. This, in general, must be determined by an interprocedural analysis. However, there are many special cases where this is very simple to do. Two such common cases were identified in this paper.

The ability to distribute computation by automatic transformations of the source programs does not, of course, guarantee improved performance; the communication time of the underlying architecture is critical. When it is large compared to the time of executing an instruction, the scope of possible applications becomes limited to those where relatively few messages are exchanged among large granules of computation. For the time being, we allow the programmer to designate which arrays should be distributed. The development of heuristics to determine when a distribution would be beneficial is subject to current research.

It is important to note that the transformation are performed on the source program. The resulting code does not contain any extensions beyond the scope of C++ as implemented on a multiprocessor, such as the Intel iPSC/2 hypercube [INT89]. Hence no modifications to the compiler are necessary; the transformations can be implemented as a preprocessor for the regular compiler.

One of the main limitations of the transformations as described in this paper is the inability of slave objects to exchange data with one another directly. This is because only the master PE knows about the distribution and thus can invoke the methods of slave objects. Hence all data must pass through the master PE. The obvious solution is to create for each slave a class similar to c_d, which knows about the distribution of other objects. This, however, would lead to deadlocks when slaves attempted to invoke each other's methods. To solve this problem in general would require that each object be capable of receiving and processing new method invocation messages while waiting for a reply from some other object. A possible solution, which involves invoking each method as a separate thread (light-weight process) is currently under investigation.

The ability to invoke methods of a given object as separate threads would also allow for multiple levels of distribution by applying the proposed distribution mechanisms recursively. Assume, for example, that some method of A[i], where A[n] is distributed, declares another (local) array of objects, B[m]. B[m] could also be distributed to all PEs in the same way as A[n]. Here the PE holding A[i] would serve as the master for this array and other PEs would be the slaves. In other words, the master/slave assignment would only be virtual, since each PE could be a master and also serve as a slave to different masters.

To avoid deadlock, however, each method would have to be invoked as a separate thread, thus allowing each PE to be suspended on more than one receive statement. This again requires a light-weight process facility in order to tolerate the context switching overhead. With only heavy-weight (Unix) processes, the distribution must be limited to a single level, performed out of a single master PE. This PE then runs the coordinating thread of computation, which may "farm out" subranges of an array of objects and then command the slaves to perform the operations on its behalf. All object declared in a slave PE remain local to that PE.

In addition to the above limitations, the following issues are also subject to current research.

Combining message to the same PE. Typically, a loop iterating over a distributed array will access all or most element of each subrange. In an environment where communication is very costly, combining messages destined to the same PE would be very beneficial. This could be achieved at the source level by providing additional methods, that would apply a given operation to (e.g., invoke a method of) all elements of the subrange held by a slave PE.

Distribution to a subset of PEs. When the array size is small, it may be necessary to distribute it to only some of the slave PE. The necessary mechanisms to achieve the transformation are relatively simple modifications of the transformations presented here. The main problem, however, is to devise heuristics to determine how many and which of the possible slave PEs should be employed. Related to this is the problem of distributing individual (non-array) objects to slave PEs.

Finally, the problem of remote parameter passing and remote function execution must be addressed. That is, the caller of a method should be able to specify an object as a parameter, which could be remote to the caller, remote to the callee, or remote to both. With this capability, data could be passed directly among slave PEs, rather than forcing it to flow through the master PE. Furthermore, functions that use remote objects as parameters and produce results used by other remote methods should execute remotely, at the site where their results will be needed. Consider for example the statement A.m(f(B)), where the result of the function f is used as a parameter for the method m. If both A and B are remote to the caller, the function f should execute at the site of A (or B), thus eliminating the flow of data through the caller.

References

[Ame87] P. America, POOL-T: A Parallel Object-Oriented Language. In *Object-Oriented Concurrent Programming*, Akinori Yonezawa and Mario Tokoro, Ed., The MIT Press, 1987, pp. 199-220.

[BLL88] B.N. Bershad, E.D. Lazowska, and H.M. Levy PRESTO: A System for Object-Oriented Parallel Programming. *Software-Practice and Experience 18*, 8 (August, 1988), pp. 713-732.

[BoLa89] J.V.D. Bos, and C. Laffra PROCOL A Parallel Object Language with Protocols. *OOPSLA '89 Proceedings* (Oct., 1989), pp. 96-102.

[DaCh88] W.J. Dally and A.A. Chien Object-Oriented Concurrent Programming in CST. *The Third Conference on Hypercube, Concurrent Computers and Applications 1* (Jan., 1988), pp. 434-439, Pasadena, CA.

[GeRo88] N.H. Gehani and W.D. Roome Concurrent C++: Concurrent Programming with Class(es). *Software—Practice and Experience 18(12)* (Dec., 1988), pp. 1157–1177.

[Gut77] J. Guttag Abstract Data Types and the Development of Data. *Comm. ACM 20, 6* (June, 1977).

[Int89] Intel Corporation. iPSC/2 User's Guide. Order# 311532-004. Intel Corporation.

[NeSn87] P.A. Nelson and L. Snyder Programming Paradigms for Nonshared Memory POarallel Computers. In *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D.B. Gannon, and R.J Douglas, Ed., The MIT Press, Cambridge, MA, 1987.

[PoLe89] C. Porter, W.J. Leddy C++ in the ES-Kit Environment. *Proc. Fourth Conf. Hypercubes, Concurrent Computers and Applications*, Monterey, CA (March, 1989).

[Str87] B. Stroustrup *The C++ Programming Language*, Addison-Wesley, 1987.

[YBU90a] M-L. Yin, L. Bic, and T. Ungerer Parallel C++ Programming on the Intel iPSC/2 Hypercube. *Fourth Annual Symposium on Parallel Processing* (April 4-6, 1990), Fullerton, CA.

[YBU90B] M-L. Yin, L. Bic, and T. Ungerer Translating C++ Programming for the Intel iPSC/2 Hypercube. *TOOLS PACIFIC'90* (Nov., 1990),.

[YoTo87] Y. Yokote and M. Tokoro Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, Akinori Yonezawa and Mario Tokoro, Ed., The MIT Press, 1987, pp. 129-158.