Parallelism Encapsulation in C++

Youfeng Wu
Ted G. Lewis
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3902

89-60-3

Parallelism Encapsulation in C++

Youfeng Wu[1]
Ted G. Lewis[2]

[1] Sequent Computer Systems, Inc.
[2] Oregon Advanced Computing Institute (OACIS)
wu@sequent.com
lewis@mist.cs.orst.edu

abstract

Object oriented programming features information hiding and encapsulation, meaning that 1) each object hides the the implementation details from access from outside and only a set of methods (interface routines) are visible outside of the object, and 2) changes to the implementation of the object do not require changes to the code that uses the object, so long as the interface is stable. However, the interface mechanism in C++ is not adequate to achieve information hiding and encapsulation when writing parallel C++ programs, since the methods are assumed to be invoked in sequence and no parallel interactions are represented by them. Also, even when the methods are the same, changes to the implementation details of the methods often affect the interaction pattern of the methods so the parallel code that uses the methods must be rewitten. To achieve information hiding and encapsulation, we propose adding path expressions to the class interface. Thus either dynamic or automatic parallelization can be used to achieve parallelism encapsulation. A new concept of data dependence analysis is introduced which uses the parallelism described by path expressions to efficiently and automatically parallelize an object-oriented program.

1. Introduction.

Features of object-oriented programming ([COX-86], [STROUSTRUP-88a]), such as information hiding and encapsulation appear to make the implementation of large systems more feasible and easier to maintain. Well defined interfaces and limited side effect reduce the chance of programing errors and encourage re -using of code. Encapsulation makes change to the inside of objects transparent to how they are used and makes software evolution less painful. It is conceivable that these concepts are also helpful in designing parallel programs as parallel programs are known to be hard to design and maintain.

Parallel programs are hard to design because a programmer must consider multiple program execution threads instead of a single thread, and must take care of all possible interactions among the threads. Parallel programs are hard to maintain because numerous interactions may have been hard-wired into the code and a simple change may affect the interaction pattern and result in global modification. We would like to extend the concept of information hiding to reduce the number of possible interactions that have to be considered, and to extend the concept of encapsulation to minimize the maintenance effort when changes are made to existing software.

The mechanisms used in object oriented programming languages for information hiding and encapsulation are not adequate in parallel programming, since the methods (interface routines) are assumed to be invoked in sequence and no parallel interactions are represented by them. Even when the methods remain unchanged, changes to the implementation details often affect the interaction pattern of the methods and the parallel code that uses the methods must be rewitten.

Among the object-oriented programming languages, C++ ([STROUSTRUP-86], [WIENDE-88]) is the most widely used object-oriented language in system programming. But there is no parallel programming support for C++ that preserves information hiding and encapsulation (e.g. Presto, [BERSHAD-88], is not compatible with the rules of information hiding and encapsulation). In this study, we propose a high level parallel programming approach in which parallelism is encapsulated within objects and thus leads to easier design and better maintainability.

2. Object-oriented Programming.

Large scale programming is by nature incremental, meaning that a larger program is built from lower level components, and these components are in turn comprised of even lower level components, and so on. In the terminology of C++ (or other object oriented programming languages), these components are called objects. However, for a component to be an object it must follow three disciplines in its design and usage. First, each object appears to the rest of the world as a few methods that can access the data of the object and the physical appearance of the data are hidden (information hiding). Secondly, only through the methods can one access the object (encapsulation). Third, objects can be organized hierarchically, with children level objects inheriting the properties of the parent level objects (inheritance). We can call the methods that use objects *subjects*, although they may be methods in other objects.

As an example, consider a program for matrix multiplication. The main body of the program inputs two matrices and computes a third matrix which is the product of the two input matrices.

```
const N=10;
main()
{
        Matrix M1(N);
        Matrix M2(N);
        Matrix M3(N);

        /* input M1, M2 */
        M3 = M1.multiply(&M2);

        /* print M3 */
}
```

The main program uses three objects which are of class Matrix, and uses the multiply method of a Matrix object:

```
Class Matrix
{
      vector *mat;
      numv n;
      transpose(); /* transpose m */

      public
            Matrix(int n); /* constructor */
            &Matrix multiply(&Matrix m);
            operator[];
            ...
}

&Matrix
Matrix::multiply(&Matrix m)
{
      int i, j;
      Matrix mtemp(m.numv);

      m.transpose();  /* transpose matrix m */
      for (i=1; i<=numv; i++)
            for (j=1; j<=numv; j++)
                  mtemp[i][j] = mat[i].innerProd(m[j]);
      return &mtemp;
}
```

Each object Matrix is implemented as an array of vector objects and matrix multiplication is implemented using the vector method innerProd:

```
Class Vector
{
      real *vec;
      int numelms;
public
      Vector(int n); /* constructor */
      operator[];
      real innerProd(&Vector v);
      &Vector sum(&Vector v);
      reverse();
```

```
        ...
}

real
Vector::innerProd(&Vector  v)
{
        int  i;
        real temp = 0.0;

        for  (i=1; i<=numelms;  i++)
                temp  =  temp  +  v[i]*vec[i];
        return  tmp;
}
```

In this example, we see that at the top level, only the method multiply (of matrix object) is used and no detail about its implementation is important. Similarly, when implementing the method multiply, the method innerProd of vector object is used, without concern for implementation details.

3. Existing Method for Specifying Parallelism in C++ Programs.

Typically, parallelism inside a C++ program is specified by inserting parallel primitives. For example, the following code implements a parallel version of matrix multiplication using Presto library objects Thread, Condition, and Monitor.

```
&Matrix
Matrix::multiply(&Matrix  m)
{
        int  i,  j;
        Matrix  mtemp(m.numv);
        Monitor alldonemon  =  new  Monitor("any");
        Condition  alldone  =  new  Condition(alldonemon,  "waiting");

        m.transpose();   /*  transpose  matrix  m  */

        /*  nThreads  is  a  new  data  in  Matrix  */
        this->nThreads = numv  *  numv;
        for (i=1; i<=numv; i++)
            for (j=1; j<=m.numv; j++) {
                Thread *t = new Thread("mul", i*numv+j,STKSZ);
                t->start(  this,
                           mat[i].innerProd, /* method */
                           m[j],             /* parameter */
                           mtemp[i,j],       /* result */
                           alldone);         /* a monitor */

        while (this->nThread) alldone->wait();

        return &mtemp;
}
```

In the above, all of the invocations of innerProd() are done in parallel. This is achieved through creating and starting a thread in place of calling an innerProd. Since all threads must be finished before the resultant matrix can be returned, busy-waiting is accomplished using condition and monitor so that only when all of the threads have finished the resultant matrix is returned. Note that in Presto, a condition object contains a monitor. The monitor controls exclusive access to the methods in the condition object (such as create, wait, signal, etc.).

The parallel solution above heavily depends on the understanding of implementation of the method innerProd. For example, for the parallel program to work correctly, it is essential that different invocations of innerProd operate independently. Also, innerProd must decrement variable nThread before it finishes its job, or the multiply method will wait forever. Furthermore, if the implementation of innerProd changes to an implementation strategy that makes different invocations of innerPord dependent, then all code that use method innerProd must be modified. These effectively break the golden rules of information hiding and encapsulation.

Parallelizing a subject by executing in parallel the methods of the lower level objects usually requires knowledge about how the methods interact. Without a systematic mechanism to abstract parallelism and hide the low level interaction details, the designer has no choice except to break the rule of information hiding, or refrain from using lower level objects by put everything in a single object. In fact, the parallel solution for matrix multiplication provided by Presto [BERSHAD-88, p713] includes innerProd as a private method in the Matrix object. This is necessary to simplify the communication between the subject and objects (modify and check state variable nThread) and also necessary to preserve object-oriented programming principles. But a Matrix object is certainly a wrong place to consider vector operations, and when a vector class is already available it is unwise to have to use primitive objects such as integer and real. Since the above parallel programming is done at the subject level without any knowledge of parallel interaction among the objects it uses, we call this approach "subject centered".

4. Path Expression and Information Hiding.

What we need is a way to specify the allowed parallelism among the methods inside each object and define the allowed parallelism in the interface to the outside world. This may require a little additional work by the designer of the object, but this is negligible since the designer has all the knowledge of the details of the object. Also, by forcing the designer to specify the interaction among the methods within an object, usually a better and cleaner design will result. Another side benefit is that the specification need be done only once and used repeatedly to save the time of all programmers who use this object.

Path Expressions [CAMBELL-74] have been studied extensively in the literature to abstractly specify synchronization among parallel activities. The typical use of a path expression is in the explicit parallel program languages ([LAUER-79], [KOLSTAD-80]), in which constructs are provided for designing "processes" that run in parallel. Path expressions are used to constrain the parallel activities. For example, the following code specifies a parallel program in which two processes cycle through operations (a; b) and (c; b) respectively. The path expression indicates that an invocation of operation b must be preceded by an operation a. The scenario in Figure 1 (a) is not allowed since the second operation b cannot execute before operation a executes. The path expression forces the scenario as in Figure 1 (b).

```
begin
        path a; b end;
P1:     process a; c end;
P2:     process c; b end;
end;
```

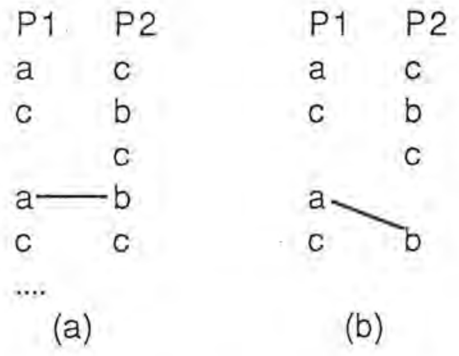

Figure 1. Process Synchronization.

Since the purpose of path expressions is to constrain parallel activities, they usually impose sequencing instead of indicating parallelism, such as "a must precede b", "a and b must not run in parallel", etc. For purposes of parallelism encapsulation, we are more interested in specifying parallelism, such as "a can run in parallel with b", "unlimited instances of a can be run in parallel", etc.

Assume a, b, c, ... is a set of methods defined in an object. The parallelism among the methods can be defined using extended path expressions ([HABERMANN-75]):

1. A method by itself is a path expression.
2. If e, e1, and e2 are path expressions, then the following are path expressions:

Notations	Meanings
e1 , e2	e1 and e2 can be run in parallel
{e}	0 or more e in parallel
e1 + e2	e1 and e2 must not run in parallel (e1 and e2 are mutually exclusive)

As an example, we take a look at the Vector class, which includes innerProd(), sum(), a reverse(), and [ ] methods. Suppose in our implementation, innerProd() and sum() do not alter the private data, and thus they can be done in parallel. On the other hand, reverse() changes the private data (method reverse() converts a vector (x1, x2, ..., xn) to a vector (xn, xn-1, ..., x1)) and it can not be executed while the other methods are going on. Thus we can specify the following path expression among the methods.

```
{innerProd, sum, [ ]} + reverse
```

As a syntactical addition to class definition, a path expression such as shown above must be given in the public area, and enclosed within key words PATH and END. The Vector class with a path expression is shown below.

```
Class Vector
{
        real *vec;
        int numelms;
public
        PATH {innerProd, sum, [ ]} + reverse END;
        Vector(int n); /* constructor */
        operator[ ];
        real innerProd(&Vector v);
        &Vector sum(&Vector v);
        reverse();
        ...
}
```

A path expression in a class describes the interactions among the methods in the class. When the methods are invoked in parallel, only the interaction patterns compatible with the path expression are allowed. Note that not all of the methods have to be included in a path expression. When a method (p) is omitted, it is assumed that the method is executed mutually exclusive of all of the other methods (p + others). If a class does not have a path expression at all, the default assumption is that all of the methods are mutually exclusive of one another.

Although a path expression is defined in a class, it specifies the parallelism among the methods that are associated with individual objects. Methods that are of the same name but associated with different objects are considered independent (except when the class has static data. In this case, a method can be prefixed by key word CLASSWISE indicating that the method in different objects conflict each other). For example, if we have two Vector objects V1 and V2. V1.reverse() only conflicts with V1.innerProd(), but not with V2.innerProd().

Note that there are two levels of parallelism we are considering: that within an object, and that within a subject that uses objects. A path expression describes only the parallelism within an object. The parallelism existing in a subject is limited to the parallelism allowed by path expressions, possibly less if the subject lets one method use the result of another method.

Parallelism specified by path expressions, can be used by programmers to write parallel subjects, or by automatic tools to convert sequential programs to parallel programs. Explicitly using the parallelism specified in path expressions to write parallel programs is not recommended, because the parallelism described in path expressions tends to change when implementation of the object changes. In other words, the interactions described in path expression is not as stable as the method interfaces over software evolution. Explicit use of the parallelism may require that the parallel program be frequently modified. So, path expressions only provide the mechanism for parallel information hiding and not parallelism encapsulation. Parallelism encapsulation requires that changes to the parallelism within an object should not affect the rest of the world that uses the object. There are two ways to achieve parallelism encapsulation: dynamic parallelization and automatic parallelization.

5. Parallelism Encapsulation through Dynamic Parallelization.

Dynamic parallelization was used in Path Pascal ([CAMBELL-80]) to achieve parallelism encapsulation. At compile time, each path expression is converted into a control engine. At runtime whenever an object is created, a control engine is implicitly created for it. Every invocation of a method is passed through the control engine, which may grant the invocation, or delay it, depending on the current state of the control engine. Also, every termination of a method must go through the control engine to update the state of the control engine.

For example, assume an object has the following methods and path expression:

```
Class Sample
{       public
        PATH {a} + b END;
        a();
        b();
}
```

The path expression PATH {a} + b END can be translated to a control engine consisting of the following two procedures, start and depart, and the two state variables, #a for the number of active operations a and #b for the number of active operations b:

```
int  start(operation)
{
        case operation {
        a: if (#b == 0)
                #a++; start a; return 1
          else
                put a in waiting queue; return 0;
          break;
        b: if (#a = 0 && #b = 0)
                #b++; start b; return 1
```

```
            else
                put b in waiting queue; return 0;
            break;
    }       }

    int  depart(operation)
    {
            case operation {
                    a: #a--; break;
                    b: #b--; break;
            }
            if (waiting queue not empty) {
                    op = dequeue(waiting queue);
                    while (start(op) && ! empty(waiting queue))
                            op = dequeue(waiting queue);
    }       }
```

The state variables #a and #b are added to the private data of the object and the procedures *start* and *depart* are two private methods of every object of class Sample. For the control engine to work correctly, calls to start and depart must be serialized (such serialization mechanism is not mentioned in the above code). Every method of a Sample object invokes the start method as its first operation and invokes the depart method as its last operation.

With dynamic parallelization, parallelism is completely encapsulated inside the object, and any change to the path expression of an object requires only a recompilation of the object (recoding the control engine). However this method assumes that there are parallel processes issuing parallel operations to the control engine. In C++ we don't have language constructs for specifying processes, but we can use the following method to create parallel processes.

Asynchronous Light Weight Processes.

Under the assumption that process creation and termination are cheap, we can convert each call to a method to an activation of a (light weight) process to execute the method. If the results of its execution (function result or new values of reference parameters) are used in a later computation, an explicit wait call is issued to delay the computation until the process terminates. Other than this kind of result-use relation, all processes (calls to methods) are assumed independent and their interactions are controlled by the control engines of the objects. For example, in the following code (a), the result of variable m is not used until the return statement. So all of the calls to method innerProd can be activated as processes that run in parallel, as in (b).

```
(a)   for (i=1; i<=N; i++)
            m[i][j] = mat1[i].innerProd(mat2[i]);
      return m;

(b)   for (i=1; i<=N; i++)
            pid[i] = activate(m[i][j],mat1[i].innerProd,mat2[i]);
      for (i=1; i<=N; i++)
            wait(pid[i]);
      return m;
```

To improve the parallelism in the method, compiler techniques can be used to advance calls and delay access to the result.

A non-trivial problem is that when a statement needs the result of a procedure call it must decide which process(es) to wait for returned data. A simple way is to count the total number of activate processes, and when a result is wanted, wait until all processes finish. More selectively issuing waits can improve parallelism.

Although the above method is simple, a major problem is that the required dynamic conflict resolution introduces runtime overhead. First the cost of executing the control engine may be more than the time saving through parallel execution, especially when the methods are small. Secondly, the control engine always has to consider all methods that might be involved in the conflict, although in reality only a few methods may be used in a particular program. The control engine may become the bottleneck for the parallel execution.

6. Parallelism Encapsulation through Automatic Parallelization.

Another approach for parallelism encapsulation is *Automatic Parallelization*. Given a C++ program (or a subject) that uses the objects whose parallelism are described by path expressions, a restructuring tool converts the sequential program to a parallel one by consulting the parallelism described in the path expressions. When the parallel pattern in an object changes, only recompilation of the parts of the program that uses the object need be done to ensure correct parallelization of the program, without any global code modification by the programmer.

Parallelism encapsulation through automatic parallelization is interesting because it takes advantage of both explicit parallelization (path expression) and implicit restructuring technology. Without parallelization tools parallel programming is too tedious ([APPELBE-85], [LUBECK-85]). On the other hand, achievable parallelism is very limited without explicit parallel programming effort ([WOLFE-87], [LEE-85]). Specifying a path expression is relatively easy since it is only necessary to consider object-local parallel activities. Still, we anticipate that highly parallel objects will result in highly parallel programs when using objects properly, although statistics are needed to support this expectation. When path expressions have been specified, restructuring becomes much easier (to be discussed soon). Parallelism encapsulation through automatic parallelization offers a natural combination of explicit and implicit parallel programming efforts.

Restructuring is centered around data dependence analysis ([KUCK-81]). Normally, to analyze whether or not two statements are dependent, the sets of used variables U and modified variables M are first determined. Two statements S1 and S2 are dependent if S1.U * S2.M, or S1.M * S2.U, or S1.M * S2.M is not empty. To analyze data dependence among the statements that involve procedure calls, interprocedural analysis is required to find the summary information, namely the sets of variables that may be used or modified by the procedure ([BART-78], [COOPER-88], [LI-88]). Interprocedual analysis is hard with little reward because procedural side effect forces the analysis to make conservative assumptions and often the summary information is far from precise at the site it is used.

In C++, all accesses to data take the form of procedure (method) calls. This implies that analyzing any data dependence requires interprocedural analysis which we know is difficult and gives no precise data dependence information. However, once parallelism has been described by path expressions, the side effect of calling the methods of an object is limited to the object itself and all the methods that are affected by the side effect have their interaction specified in the path expression. So, if calls to methods take no reference parameters (these are the dominating usage of methods: sending messages to methods), path expressions encapsulate completely the information about the interactions among methods and data dependence analysis is trivial. If calls do take reference parameters, we need additional analysis to find the set of indirectly called methods of the parameter objects. This analysis resembles normal interprocedural analysis, but it is simpler since no global variables need to be considered ([COOPER -88]). We can analyze data dependence for C++ programs using path expressions as follows.

Data Dependence Analysis using Path Expression.

The first step to perform data dependence analysis is to associate each statement with a set C, the set of methods that are called in the statement.

The methods that are called in a statement come in two forms. The usual form is the direct call, such as v.reverse(). Note that an assignment to an object is also a call to the (overloaded) assignment method of the object, such as v = something, which is equal to v.=(something). Another form of calling a method is the indirect call by passing an object as a reference parameter to another method and calling the methods of the parameter inside the called method. An example of indirect call is that m.[] is indirectly called inside mat.multiply(m).

Once we have determined the set of methods S.C called by every statement S, statements S1 and S2 are data dependent if and only if there exists a call obj1.method1() in S1.C and a call obj2.method2() in S2.C, such that

1. obj1 = obj2 and
2. (method1, method2) is not compatible with the path expression in obj1.

For example, in the following,

```
S1:   r1 = v.innerProd(v);
S2:   r2 = v1.innerProd(v2);
S3:   v1.reverse();
S4:   v2.reverse();
```

S1.C = (r1.=, v.innerProd, v.[]), S2.C = (r2.=, v1.innerProd, v2.[]), S3.C = (v1.reverse), and S4.C = (v2.reverse). Statements S1 and S2 are independent since they call methods of different objects. S2 and S3 are dependent because they call methods in the same object and (innerProd, reverse) is not compatible with the path expression for vector object. S2 and S4 are also dependent since object v2.[] is called in S2 and v2.reverse is called in S4 and ([], reverse) is not compatible with the path expression for object vector.

Similarly, two statements S1, S2 have loop carried dependence ([ALLEN-83]) if and only if there are two iteration vectors I= (i1, i2, ..., in), J= (j1, j2, ..., jn) and calls obj1.method1 in S1(I).C, obj2.method2 in S2(J).C, such that

1. $I \neq J$;
2. obj1 = obj2;
3. {method1, method2} is not compatible with the path expression of obj1.

As another example, in the following loop, statements S1 and S2 have no loop carried dependence since they call methods in different objects. However, S2 and S3 have loop carried dependence because iteration vector (i) ≠ (i+1), method v1[i+1].reverse() is called by S2(i+1), method v1[i+1].innerProd() is called by S3(i), and {innerProd, reverse} is not compatible with the path expression for object vector.

```
for (i=1; i<=n; i++) {
        S1:   r1[i] = v[i].innerProd(v[i]);
        S2:   v1[i].reverse();
        S3:   r2[i] = v1[i+1].innerProd(v2);
}
```

Determining Compatible Path Expressions.

To check whether or not a path expression e2 is compatible with another path expression e1, we need the following concepts.

1) The *projection* of path expression e1 to e2, projection(e1, e2), is the path expression obtained by removing the methods in e1 that are not in e2.

2) The *standard* form of a path expression e, standard(e), is the path expression obtained after re-arranging e so that the operands of the commutative operators "+", and "," are in a specific order (e.g. the lexical order of the operands).

3) Assume e1, e2, e3, e4 are path expressions. The relation $\leq_S$ (*Strictly compatible* relation) on the set of path expressions can be defined as follows.

. If e1 = e2, then e1 $\leq_S$ e2.

. If e1 $\leq_S$ e2 and e3 $\leq_S$ e4, then
(e1 , e3) $\leq_S$ (e2 , e4);
(e1 + e3) $\leq_S$ (e2 + e4);
{e1} $\leq_S$ {e2}.

. If e1 $\leq_S$ e2, then e1 $\leq_S$ {e2}.

Path expression e2 is compatible with path expression e1 if and only if standard(e2) $\leq_S$ standard(projection(e1, e2)).

For example, assume e1 = {innerProd, sum, [ ]} + reverse, and e2

= (sum, innerProd). Since standard(e2) = (innerProd, sum) $\leq_s$ {innerProd, sum} = standard(projection(e1, e2)), we conclude that e2 is compatible with e1. On the other hand, e2 = {innerProd, reverse} is not compatible with e1 = {innerProd, sum, [ ]} + reverse, since projection(e1, e2) = {innerProd} + reverse, whose standard form is not strictly compatible with standard({innerProd, reverse}).

Determining Indirect Calls.

The problem of determining the set of indirect calls in method p(V) is to find the set of methods that are associated with the reference parameters in V and are called inside p. We don't need to consider the methods of the value parameters because the methods of the value parameters do not conflict with any other methods at the call site. So we can assume V contains only reference parameters.

Assume method p has formal parameters FV. We fomulate the following flow problem for the set of indirect calls of p(V).

. For the set of reference formal parameters FV of p, we denote the set of calls to FV's methods directly issued inside p as DC(p, FV).

. For each nested call qi, let FVi be the set of formal parameters of p passed to it (FV $\supseteq$ FVi). We denote the set of indirect calls of qi(FVi) as IDC(qi, FVi).

Then the methods indirectly called by p(FV) are

$$\mathrm{IDC}(p, FV) = \mathrm{DC}(p, FV) \cup \bigcup_{\forall q_i \text{ called by p with } FV_i} \mathrm{IDC}(q_i, FV_i)$$

The methods indirectly called by p(V) are IDC(p, FV) with FV replaced by the corresponding objects in V.

Note that a method indirectly called by p(V) does not have to be a method of any object in V. For example, in the following, method obj1.[](1).= is indirectly called which is a method of object returned by the call to obj1.[](1). Similarly obj2.subm(x, y).= is indirectly called even though it is neither a method of obj1 nor a method of obj2.

```
mtd(&OBJECT obj1, obj2);
{
        obj1[1] = data1;
        obj2.subm(obj1, y) = data2;
}
```

The above example implies that an indirectly called method can be a variable. Thus sometime it is impossible to determine the precise set of indirectly called methods. When this happens, we have to conservatively assume a large set of calls covering the uncertainty.

Relation to Normal Data Dependence Analysis.

In a non-object oriented programming language, accesses to a variable are classified as either using or modifying and the two accesses are considered in conflict when they are performed in parallel with the same data (we do not distinguish between the false dependences and the true dependence [KUCK-81] since most false dependences can be removed by renaming and using temporary variables). In C++, accesses to objects are no longer either using or modifying. Instead, every method is a unique type of access to an object. Whether or not the methods are in conflict is described in the path expression. So normal data dependence analysis is a more primitive form of data dependence analysis using path expressions. In particular, data dependence analysis using path expressions can be used to simulate normal data dependence as follows.

Consider every variable in a non-object oriented program as belonging to the same univeral class UNIVERSE with two public methods use() and modify(), and with path expression PATH use + modify END. Each use of a variable x is a call to x.use(), and each modification of variable x is a call to x.mod(). Then for any statement S, S.C = S.U + S.M, and the result of whether statements S1 and S2 are dependent will be the same either by normal data dependence analysis or by analysis using path expressions.

The ability to simulate normal data dependence analysis is of particular interest because C++ can be used as a non-object oriented "better" C ([STROUSTRUP-88b], [WIENER-88]), as well as an object oriented programming language. When a C++ program is written including non-object oriented features, such as free variables (the variables that are not encapsulated inside an object) and free functions (the functions that are not methods of any object), the same data dependence analysis frame work can be used by simply assuming that all free variables are the objects of the universal class with methods use() and modify() and path expression PATH use + modify END. When a statement calls a free function, determining the set of indirect calls is the same as normal interprocedural analysis.

7. Conclusions and Open Problems.

In realizing that current techniques for parallel programming in C++ destroy features like information hiding and encapsulation, we propose to extend the C++ interfacing mechanism by adding path expressions to describe parallel interactions among the interface methods and to hide the details of parallel interactions from the outside. Furthermore, since parallel interactions among the interface methods tend to change when the implementation strategy changes, we propose two methods, dynamic parallelization and restructuring, to automatically propagate parallelism from the inside of objects to the code that uses the objects.

Automatic propagation of parallelism through restructuring can be done efficiently by taking advantage of the information available in path expressions. This method is shown to be more general than the normal restructuring approach, and thus the frame work presented here can be used when non-object oriented features are mixtured with objects in a single C++ program. Being able to work with non-object oreinted features is of particular importance in C++ since C++ is not a "pure" object oriented programming language.

We have used examples of the simplest form of C++ objects, namely the objects that have no public data, friend methods, derived classes, static data, virtual functions, overloaded functions, or multiple inheritance ([STROUSTRUP-86, -87]). We can show (omitted here to save space) that these present no difficulty (except dynamic binding of virtual functions) to our parallelism encapsulation strategies.

Since the operands in our path expression are functions, we are assuming large grain parallelization. It is interesting to study the possibility of extending path expressions to allow finer grain parallelism ([BRUEGGER-83]). Another open problem is posed by conditional path expression ([ANDLER-79]), since two methods may be run in parallel under certain conditions, but not always. Allowing a conditional path expression will increase the parallelism. Our method does not allow conditional parallelism.

8. References.

[ALLEN-83] Allen, J. R., Dependence Analysis for Subscripted Variables and Its Applications to Program Transformations, Ph.D. Thesis, Rice University, Houston, April 1983.

[ANDLER-79] Andler, Sten, "Predicate Path Expressions: A High-Level Synchronization Mechanism," Ph. D. Thesis, Carnegie Mellon University, Aug. 1979.

[APPELBE-85] Appelbe, W. F. and C. McDowell, "Anomaly Detection in Parallel Fortran Programs," Proc. Workshop on Parallel Processing Using the HEP, May 1985.

[BARTH-78] Barth, J.M., "A Practical Interprocedural Data Flow Analysis Algorithm," CACM 21(9), Sept. 1978.

[BERSHAD-88] Bershad, Brian N., Edward D. Lazowska and Henry M. Levy, "PRESTO: A System for Object-oriented Parallel Programming," Software-Practice and Experience, Vol. 18(8), 713-732 (Aug. 1988).

[BRUEGGER-83] Bruegger, Bernd, and Peter Hibbard, "Generalized Path Expressions: A High Level Debugging Mechanism," The Journal of System and Software 2(3), 265-276, 1983.

[CAMBELL-74] Cambell, R. H., and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," In G. Goos and J. Hartmanis (ed), Lecture Notes in Computer Science. Vol. 16, Operating Systems, pp89-102. Springer-Verlag, Berlin, 1974.

[CAMBELL-80] Cambell, R. H. and R. B. Kostad, "A Practical Implementation of Path Pascal," Tech Report, Dept of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-80-1008, 1980.

[COOPER-88] Cooper, Keith D. "Fast Interprocedural Alias Analysis," Dept. of Computer Science, Rice University, Rice COMP TR88-80, Nov. 1988.

[COX-86] Cox, Brad, "Object Oriented Programming -- An Evolutionary Approach," Addison-Wesley, 1986.

[HABERMANN-75] Habermann, A. N., "Path Expressions," Tech Report, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, June, 1975.

[KOLSTAD-80] Kolstad, Robert B. and Roy H. Cambell, "Path Pascal User Manual," ACM SIGPLAN Notices (Sept 1980), 15, 9, pp 15-25.

[KUCK-81] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Proc. 8th ACM Symp. Principles Programming Languages, Jan. 1981, pp. 207-218.

[LAUER-79] Lauer, P. E, P. R. Torrigian, and M. W. Shields, "COSY - A System Specification Language Based on Paths and Processes," Acta Informatica 12, 109-158 (1979).

[LEE-85] Lee, Gyungho, Clyde P. Kruskal, and David J. Kuck, An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors, IEEE Trans. on Computers, Vol. c-34, No. 10, October 1985.

[LI-88] Li, Zhiyuan, Pen-Chung Yew, "Interprocedural Analysis for Parallel Computing," Proc. of 1988 ICPP, Vol. II, pp 221-228.

[LUBECK-85] Lubeck, O. M., P. O. Frederickson, R. E. Hiromoto, and J. W. Moore, "Los Alamos Experiences with the HEP Computer," in Parallel MIMD Computation: HEP Supercomputer and Its Applications (MIT Press, 1985).

[STROUSTRUP-86] Stroustrup, Bjarne, "The C++ Programming Language," Addison-Wesley Publishing Company, Inc, 1986.

[STROUSTRUP-87] Stroustrup, Bjarne, "Multiple Inheritance for C++," Proceedings of the Spring' 87 BUUG Conference, Helsinki, May, 1987.

[STROUSTRUP-88a] Stroustrup, Bjarne, "What is "Object Oriented Programming"?," IEEE Software, Vol. 5 No. 3, May, 1988, pp 10-20.

[STROUSTRUP-88b] Stroustrup, Bjarne, "A Better C?," Byte Magazine, Aug. 1988, pp 215-216D.

[WIENDE-88] Wiener, Richard S., Pinson, Lewis J., "An Introduction to Object-Oriented Programming and C++," Addison-Wesley Publishing Company, Inc, 1988.

[WOLFE-87] Wolfe, M.J. and Utpal Banerjee, "Data Dependence for Parallelism Detection," Int'l Journal of Parallel Programming, Vol. 15, No. 2, April, 1987.