Matrix-Vector Product: A Small Example of Parallel Processing Using CORBA

David R. Musser

March 23, 1999

1 Introduction

In this note I describe a distributed version of a matrix-vector product computation, devised as simple demonstration of a CORBA client-server system in which the client performs an algorithm that distributes parts of a large computation to two or more servers. The servers are functionally identical, but they can be compiled with CORBA implementations on different operating systems and run on different machines from the client and from each other. With a bit more work it should be possible also to make them work with different CORBA implementations and/or with different parts implemented in different languages. In this note the omniORB CORBA implementation is used; the example has also been compiled and tested with the MICO ORB, but not with both ORBs involved in the same computations. The implementation language in all cases is C++, but in some experiments two different C++ compilers have been used; e.g., Microsoft Visual C++ 6.0 on Windows 95 and GNU C++ 2.8.1 on SunOS.

These programs are only intended as simple exercises aimed at exploring ideas about combining generic programming techniques with distributed computing. For a serious effort at providing distributed components for linear algebra, one should start with close examination of efforts such as the PETSc Library,[1] which does distributed computation but doesn't employ generic programming, or the Matrix Template Library (MTL),[2] which makes extensive use of generic programming but doesn't (yet) do distributed computation.

Besides having the flexibility to have servers and clients inter-operating on different machines or operating systems, this example provides further opportunities for generality. Varying the type of data elements is one possibility. Another, with which the term "generic programming," is perhaps most often identified, is varying the type of sequence data structure used, without having to change the algorithm(s) involved; that is, the algorithms are generic with respect to different data structures. As currently written, the program uses the vector representation from the C++ standard library, but the product algorithm would work equally well with a standard library list representation. (Actually this is not quite true with the algorithm is described here, but with small modifications we could obtain an algorithm that would work with both a list and a vector representation.)

2 Client-server structure

The distributed matrix-product system consists of:

- A server that provides a matrix object equipped with a `block_product` function, which is a function that is able to use a designated range of rows of a matrix `m` and a column vector `c` and produce the corresponding range of elements of the column

[1] http://www-unix.mcs.anl.gov/petsc
[2] http://lsc.nd.edu/research/mtl

vector `r` that is the matrix-vector product of `m` and `c`. This server registers the object reference of the matrix with the CORBA Naming Service, using a name specified by the user on the command line. One should start two copies of the server, each with a different name; e.g.,

```
server matrix1
server matrix2
```

- A client that connects to two matrix servers holding the same matrix and uses their matrix `block_product` functions. The call of the first server's `block_product` function computes the top half of a matrix-vector product and the call on the second server does the second half. These calls are done in parallel, by making the first call via the Dynamic Invocation Interface, as described later. One tells the clients the names of the servers with two parameters on the command line, and the client uses the Naming Service to look them up and obtain the corresponding object references.

3 IDL definitions

For this simple example, the entire IDL specification consists of just these two parts:[3]

"matrix.idl" 2a ≡
⟨Row and Column types 2b⟩
⟨Matrix interface 3a⟩

The interfaces of the CORBA objects used in this example are expressed in CORBA's Interface Definition Language (IDL). First we define `Row` and `Column` objects in terms of IDL's built-in `Sequence` type:

⟨Row and Column types 2b⟩ ≡

```
typedef sequence<double> Row;
typedef sequence<double> Column;
```

Used in part 2a.

The IDL `sequence` type is similar in its functionality to the C++ standard library `vector` type but is more limited. In a later section we consider the possibility of defining an IDL interface for (special cases of) the `vector` type and redoing this matrix example in terms of it.

We use `double` is used as the type of row, column, and matrix elements. CORBA has no mechanism for compile-time type parameterization analogous to C++ templates or Ada generics, but we could make this example more generic by using its dynamic type system, using type `Any` in place of `double`. This is left for a future extension.

We next define the `Matrix` type, essentially as a `sequence` of `Row`'s. Actually, we define `Matrix` not just with a `typedef` but with an `interface` definition, which allows us to provide its objects with operations we intend to provide:

[3] Throughout we present code in a variant of Knuth's literate programming style [Knuth84], in which code is presented in "parts" numbered according to the page number on which they appear (with parts on the same page distinguished by appending a letter to the number). This form of presentation is supported by Briggs' Nuweb tool [Briggs] (slightly modified) with which we also generate all code files directly from the document's source file. Parts may refer to other parts that have either already been defined or are defined later in the document; one can choose whatever order of presentation seems best for expository purposes, while the order in which code appears in the compilation files is a matter of how the parts are ordered when included in file parts such as `matrix.idl`. See also Appendix C. (Even experienced Nuweb users should see the note there about problems with including line directives in IDL files.)

⟨Matrix interface 3a⟩ ≡

```
interface Matrix {
   typedef sequence<Row> rep;
   Column block_product(in unsigned long first,
                        in unsigned long last,
                        in Column c);
};
```

Used in part 2a.

In this case we specify only one operation, block_product, which takes two parameters of type unsigned long and one of type Column; it returns an object of type Column.

The rep type is for use in the implementation, but in a larger example there might be operations that have this as a parameter type or return type.

4 Matrix server implementation

In the server program we provide an implementation of the block_product function, set up a matrix, and initialize to perform the block_product operation on the matrix at the request of clients. The overall structure of the server is as follows.

"server.cpp" 3b ≡

```
⟨Include standard and CORBA library headers 3c⟩
⟨Include matrix object interface generated by IDL compiler 3d⟩
⟨Include auxiliary file to help with using the Naming Service 9a⟩
⟨Define matrix object implementation 4e⟩

int main(int argc, char *argv[])
{
  assert(argc == 2);
  const int N = 20;
  ⟨Set up matrix m0 4a⟩
  ⟨Initialize ORB connection 4b⟩
  ⟨Use m0 to initialize a matrix object, matrix1, and serve it up 4c⟩
  ⟨Register matrix1 with the Naming Service 4d⟩
  boa->impl_is_ready();
  return 0;
}
```

For this simple example only a few header files are needed.

⟨Include standard and CORBA library headers 3c⟩ ≡

```
#include <vector>
#include <iostream.h>
using namespace std;
#pragma warning (disable : 4786)
#include <omniORB2/CORBA.h>
#include "omnithread.h"
```

Used in parts 3b, 5d.

Note that <iostream.h> is included, not the now-standard <iostream>, because the latter leads to compilation errors under VC++. This might need to be changed when using another compiler.

⟨Include matrix object interface generated by IDL compiler 3d⟩ ≡

```
#include "matrix.h"
```

Used in parts 3b, 5d.

To keep the example simple, we construct the matrix that `block_product` operates on in the server.

⟨Set up matrix m0 4a⟩ ≡

```
vector<double> r;
vecrep m0;

for (int i = 0; i < N; ++i) {
  r.erase(r.begin(), r.end());
  for (int j = 0; j < N; ++j)
    r.push_back(i*N + j);
  m0.push_back(r);
}
```

Used in part 3b.

The server initialization is standard.

⟨Initialize ORB connection 4b⟩ ≡

```
cout << "Initializing..." << endl;
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB2");
CORBA::BOA_var boa = orb->BOA_init(argc, argv, "omniORB2_BOA");
```

Used in parts 3b, 5d.

The type used in the following initialization is `Matrix_impl`, which is implemented by a class derived, as required by the omniORB object interface conventions, from the class `sk_Matrix` generated by the omniORB IDL compiler.

⟨Use m0 to initialize a matrix object, matrix1, and serve it up 4c⟩ ≡

```
Matrix_impl matrix1(m0);
matrix1._obj_is_ready(boa);
```

Used in part 3b.

⟨Register matrix1 with the Naming Service 4d⟩ ≡

```
Matrix_var myobjRef = matrix1._this();
if (!bindObjectToName(orb, myobjRef, "test", argv[1])) {
  return 1;
}
```

Used in part 3b.

⟨Define matrix object implementation 4e⟩ ≡

```
typedef vector<vector<double> > vecrep;

⟨Define inner product function 5c⟩

class Matrix_impl : virtual public _sk_Matrix {
    rep mr;
    int _calls;
public:
    ⟨Constructor, for conversion from vector representation 5a⟩
    ⟨Block product member function 5b⟩
};
```

Used in part 3b.

The constructor copies from a matrix given in vector form to a representation in terms of the CORBA sequence type.

⟨Constructor, for conversion from vector representation 5a⟩ ≡
```
Matrix_impl (const vecrep& m0)
{
  _calls = 0;
  mr.length(m0.size());
  for (int i = 0; i < m0.size(); ++i) {
    mr[i].length(m0[i].size());
    for (int j = 0; j < m0[i].size(); ++j)
      mr[i][j] = m0[i][j];
  }
}
```
Used in part 4e.

⟨Block product member function 5b⟩ ≡
```
Column* block_product(CORBA::ULong first,
                      CORBA::ULong last, const Column& c )
{
  cout << "block_product(" << first << ", " << last << ", c) start"
       << endl;
  Column* z = new Column;
  z->length(last - first);
  int k = 0;
  for (int i = first; i != last; ++i, ++k)
    (*z)[k] = inner_product(mr[i], c);
  cout << "block_product(" << first << ", " << last << ", c) end"
       << endl;
  return z;
}
```
Used in part 4e.

Finally, the inner product function used in computing the block product is written in terms of the Row and Column types.

⟨Define inner product function 5c⟩ ≡
```
double inner_product(const Row& r, const Column& c) {
  double result = 0;
  for (int i = 0; i < r.length(); ++i)
    result += r[i] * c[i];
  return result;
}
```
Used in part 4e.

5 Matrix client

This is a client that connects to two matrix servers holding the same matrix and uses their matrix `block_product` functions. The call of the first server's `block_product` function computes the top half of a matrix-vector product and the call on the second server does the second half. These calls are done in parallel, making use of a feature of the Dynamic Invocation Interface.

`"client.cpp"` 5d ≡
```
⟨Include standard and CORBA library headers 3c⟩
⟨Include matrix object interface generated by IDL compiler 3d⟩
⟨Include auxiliary file to help with using the Naming Service 9a⟩

int main( int argc, char *argv[] )
{
  ⟨Initialize ORB connection 4b⟩
```

```
  vector<Matrix_var> client;
  ⟨Initialize client[0] and client[1] using Naming Service 6a⟩
  const int N = 20;
  ⟨Create a column of all 3's 6b⟩
  cout << "Computing the matrix-vector product..." << endl;
  vector<Column*> result(2);
  ⟨Compute top and bottom halves of matrix product in parallel 7a⟩
  ⟨Output first half of result from result[0], second half from result[1] 7g⟩
  cout << endl;
  return 0;
}
```

⟨Initialize client[0] and client[1] using Naming Service 6a⟩ ≡

```
int j;
for (j = 1; j <= 2; ++j) {
  cout << "j = " << j << endl;
  try {
    CORBA::Object_var obj = getObjectReference(orb, "test", argv[j]);
    client.push_back(Matrix::_narrow(obj));
    if (CORBA::is_nil(client[j-1])) {
      cerr << "Cannot invoke on a nil object reference.\n" << endl;
      return 1;
    }
    assert(!CORBA::is_nil(client[j - 1]));
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
    return 1;
  }
  catch(omniORB::fatalException& ex) {
  cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
       << "within omniORB2.\nPlease send a bug report.\n"
       << "The exception was thrown in file: " << ex.file() << "\n"
       << "                            line: " << ex.line() << "\n"
       << "The error message is: " << ex.errmsg() << endl;
    return 1;
  }
  catch(...) {
    cerr << "Caught a system exception." << endl;
  }
}
```

Used in part 5d.

We create (arbitrarily) a `Column` to send to the servers in the `block_product` calls.

⟨Create a column of all 3's 6b⟩ ≡

```
Column c;
c.length(N);
for (int i = 0; i < N; ++i)
  c[i] = 3;
```

Used in part 5d.

For the `block_product` calls, we want to do the equivalent of

```
result[0] = client[0]->block_product(0, N/2, c);
result[1] = client[1]->block_product(N/2, N, c);
```

except have them execute in parallel. To achieve this we do the first `block_product` using the DII and send the request with `send_deferred` instead of (the normal) invoke call. Thus, the first call does not block and we can proceed with the second `block_product` (calling it in the usual way).

⟨Compute top and bottom halves of matrix product in parallel 7a⟩ ≡
⟨Set up first half with DII request 7b⟩
⟨Asynchronously do block_product on second server 7c⟩
⟨Wait for the first block_product call to finish 7d⟩
⟨Check for exceptions 7e⟩
⟨Get the return value from the first half 7f⟩

Used in part 5d.

⟨Set up first half with DII request 7b⟩ ≡

```
CORBA::Request_var _req = client[0]->_request( "block_product" );
_req->add_in_arg( "first" ) <<= (CORBA::ULong)0;
_req->add_in_arg( "last" ) <<= (CORBA::ULong)(N/2);
_req->add_in_arg( "c" ) <<= c;
_req->set_return_type(_tc_Column);
_req->send_deferred();
```

Used in part 7a.

⟨Asynchronously do block_product on second server 7c⟩ ≡

```
result[1] = client[1]->block_product(N/2, N, c);
```

Used in part 7a.

⟨Wait for the first block_product call to finish 7d⟩ ≡

```
_req->get_response();
```

Used in part 7a.

⟨Check for exceptions 7e⟩ ≡

```
if(_req->env()->exception()) {
  CORBA::Exception *_ex = _req->env()->exception();
  /* Following is from MICO version, needs to be ported to omniORB
  CORBA::UnknownUserException *_uuex =
     CORBA::UnknownUserException::_downcast( _ex );
  if(_uuex) {
    mico_throw( CORBA::UNKNOWN() );
  } else {
    mico_throw( *_ex );
  }
  */
}
```

Used in part 7a.

⟨Get the return value from the first half 7f⟩ ≡

```
_req->return_value() >>= result[0];
```

Used in part 7a.

⟨Output first half of result from result[0], second half from result[1] 7g⟩ ≡

```
cout << "The result: " << endl;
for (j = 0; j < 2; ++j) {
  for (int k = 0; k < result[j]->length(); ++k)
    cout << (*result[j])[k] << " ";
}
```

Used in part 5d.

6 Running the servers and client

When the client and two instances of the server are run as described in the second section, the output by the client is as follows:

⟨Client output 8a⟩ ≡
```
D:\dsc\omniorb\examples\matrix2>client matrix1 matrix2
Initializing...
j = 1
j = 2
Computing the matrix-vector product...
The result:
570 1770 2970 4170 5370 6570 7770 8970 10170 11370 12570 13770 14970
16170 17370 18570 19770 20970 22170 23370
```
Not used.

(The last line was manually broken to fit within the margins.)

The output by the servers is

⟨Server 1 output 8b⟩ ≡
```
D:\dsc\omniorb\examples\matrix2>server matrix1
Initializing...
block_product(0, 10, c) start
block_product(0, 10, c) end
```
Not used.

⟨Server 2 output 8c⟩ ≡
```
D:\dsc\omniorb\examples\matrix2>server matrix2
Initializing...
block_product(10, 20, c) start
block_product(10, 20, c) end
```
Not used.

References

[Briggs] P. Briggs, *Nuweb, a simple literate programming tool*, Version 0.87, 1989.

[Knuth84] D.E. Knuth, Literate programming. *Computer Journal* 27 (1984), 97–111.

A Name binding auxiliary functions

The functions `bindObjectToName` and `getObjectReference`, used in the server and client respectively, are used to access the Naming Service. These functions, which are provided in the auxiliary file `namebind.cpp`, are based on the Echo example that comes with OmniORB.

⟨Include auxiliary file to help with using the Naming Service 9a⟩ ≡

```
#include "namebind.cpp"
```

Used in parts 3b, 5d.

"namebind.cpp" 9b ≡

```
static CORBA::Boolean
  bindObjectToName(CORBA::ORB_ptr orb, CORBA::Object_ptr obj,
                   const char* context, const char* name)
{
  CosNaming::NamingContext_var rootContext;

  try {
    // Obtain a reference to the root context of the Name service:
    CORBA::Object_var initServ;
    initServ = orb->resolve_initial_references("NameService");

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    cerr << "Service required is invalid [does not exist]." << endl;
    return 0;
  }

  try {
    CosNaming::Name contextName;
    contextName.length(1);
    contextName[0].id   = (const char*) context;
    contextName[0].kind = (const char*) "my_context";
    // Note on kind: The kind field is used to indicate the type
    // of the object. This is to avoid conventions such as that used
    // by files (name.type -- e.g. test.ps = postscript etc.)

    CosNaming::NamingContext_var testContext;
    try {
      // Bind the context to root, and assign testContext to it:
      testContext = rootContext->bind_new_context(contextName);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      // If the context already exists, this exception will be raised.
      // In this case, just resolve the name and assign testContext
      // to the object returned:
      CORBA::Object_var tmpobj;
      tmpobj = rootContext->resolve(contextName);
      testContext = CosNaming::NamingContext::_narrow(tmpobj);
      if (CORBA::is_nil(testContext)) {
```

```
          cerr << "Failed to narrow naming context." << endl;
          return 0;
        }
      }

      // Bind the object (obj) to testContext, naming it FrontOffice;
      CosNaming::Name objectName;
      objectName.length(1);
      objectName[0].id   = (const char*) name;
      objectName[0].kind = (const char*) "Object";


      // Bind obj with name objectName to the testContext:
      try {
        testContext->bind(objectName,obj);
      }
      catch(CosNaming::NamingContext::AlreadyBound& ex) {
        testContext->rebind(objectName,obj);
      }
      // Note: Using rebind() will overwrite any Object previously bound
      //       to /test/FrontOffice with obj.
      //       Alternatively, bind() can be used, which will raise a
      //       CosNaming::NamingContext::AlreadyBound exception if the name
      //       supplied is already bound to an object.

      // Amendment: When using OrbixNames, it is necessary to first try bind
      // and then rebind, as rebind on its own will throw a NotFoundexception if
      // the Name has not already been bound. [This is incorrect behaviour -
      // it should just bind].
    }
    catch (CORBA::COMM_FAILURE& ex) {
      cerr << "Caught system exception COMM_FAILURE, unable to contact the "
           << "naming service." << endl;
      return 0;
    }
    catch (omniORB::fatalException& ex) {
      throw;
    }
    catch (...) {
      cerr << "Caught a system exception while using the naming service."
           << endl;
      return 0;
    }
    return 1;
  }
```

File defined by parts 9b, 10.

"namebind.cpp" 10 ≡

```
  static
  CORBA::Object_ptr
    getObjectReference(CORBA::ORB_ptr orb, const char* context,
                       const char* name)
  {
    CosNaming::NamingContext_var rootContext;

    try {
      // Obtain a reference to the root context of the Name service:
      CORBA::Object_var initServ;
```

```
      initServ = orb->resolve_initial_references("NameService");

      // Narrow the object returned by resolve_initial_references()
      // to a CosNaming::NamingContext object:
      rootContext = CosNaming::NamingContext::_narrow(initServ);
      if (CORBA::is_nil(rootContext))
        {
          cerr << "Failed to narrow naming context." << endl;
          return CORBA::Object::_nil();
        }
    }
    catch(CORBA::ORB::InvalidName& ex) {
      cerr << "Service required is invalid [does not exist]." << endl;
      return CORBA::Object::_nil();
    }

    CosNaming::Name fullName;
    fullName.length(2);

    fullName[0].id   = (const char*) context;
    fullName[0].kind = (const char*) "my_context";
    fullName[1].id   = (const char*) name;
    fullName[1].kind = (const char*) "Object";
    // Note on kind: The kind field is used to indicate the type
    // of the object. This is to avoid conventions such as that used
    // by files (name.type -- e.g. test.ps = postscript etc.)


    CORBA::Object_ptr obj;
    try {
      // Resolve the name to an object reference, and assign the reference
      // returned to a CORBA::Object:
      obj = rootContext->resolve(fullName);
    }
    catch(CosNaming::NamingContext::NotFound& ex)
      {
        // This exception is thrown if any of the components of the
        // path [contexts or the object] aren't found:
        cerr << "Context not found." << endl;
        return CORBA::Object::_nil();
      }
    catch (CORBA::COMM_FAILURE& ex) {
      cerr << "Caught system exception COMM_FAILURE, unable to contact the "
           << "naming service." << endl;
      return CORBA::Object::_nil();
    }
    catch(omniORB::fatalException& ex) {
      throw;
    }
    catch (...) {
      cerr << "Caught a system exception while using the naming service."
           << endl;
      return CORBA::Object::_nil();
    }
    return obj;
  }
```

File defined by parts 9b, 10.

B A Makefile

This Makefile is for use with the Microsoft Visual C++ compiler. Note the IDLFLAGS variable, which defines the flags necessary to cause the omniORB IDL compiler to include certain type information needed for DII calls. These are specific to omniORB. Some of the file paths used in other variables are specific to the way omniORB is installed.

"Makefile" 12 ≡

```
# Filename   : Makefile
# Adapted from one written by : Stephen Cerniglia
# Creation Date : February 21, 1999

# Compiler Macros
IDL      = omniidl2
IDLFLAGS = -a -h .h -s Skel.cpp
CPP      = cl.exe
OMNIDEFS = /D__WIN32__ /D__x86__ /D__NT__ /D__OSVERSION__=4
CPPFLAGS = /c /O2 /MD /GX  $(OMNIDEFS)
LD       = link.exe
#LDFLAGS  = -nologo
#LDFLAGS  = /nologo /debug /subsystem:console /incremental:no

# Project Macros
PROJECT = matrix
SERVER  = server
CLIENT  = client
LIBDIR  = d:\dsc\omniorb\omniorb_2.7.0\lib\x86_win32
OMNILIBS = omniDynamic270_rt.lib omniORB270_rt.lib \
   omnithread2_rt.lib wsock32.lib advapi32.lib \
   -libpath:d:\dsc\omniorb\omniORB_2.7.0\lib\x86_win32
INCLDIR = /ID:\dsc\omniorb\OmniORB_2.7.0\include


all: idl both server client

both: nuweb
        $(CPP) $(INCLDIR) $(CPPFLAGS) $(PROJECT)Skel.cpp \
              /Fo$(PROJECT).obj
        $(CPP) $(INCLDIR) $(CPPFLAGS) $(PROJECT)DynSkel.cpp \
              /Fo$(PROJECT)Dyn.obj

server: nuweb
        $(CPP) $(INCLDIR) $(CPPFLAGS) $(SERVER).cpp \
              /Fo$(SERVER).obj
        $(LD) $(LDFLAGS) -out:$(SERVER).exe $(OMNILIBS) \
              $(PROJECT).obj $(SERVER).obj

client: nuweb
        $(CPP) $(INCLDIR) $(CPPFLAGS) $(CLIENT).cpp \
              /Fo$(CLIENT).obj
        $(LD) $(LDFLAGS) -out:$(CLIENT).exe $(OMNILIBS) \
              $(PROJECT).obj $(PROJECT)Dyn.obj $(CLIENT).obj

idl: nuweb
        $(IDL) $(IDLFLAGS) $(PROJECT).idl

zip: nuweb
        zip all *.cpp *.h *.idl Makefile
```

```
clean:
	del *.obj
	del *.exe
	del $(PROJECT)Skel.cpp
	del $(PROJECT).h
	del *.pdb

nuweb: matrix.w
	nuweb matrix

doc: nuweb
	latex matrix
	nuweb matrix
	latex matrix
	dvips -o matrix.ps matrix
```

C Hints for using Nuweb

The Nuweb tool used in this exercise is a minor modification to Brigg's [Briggs] original tool, or actually to a version previously modified by Ramsdell and Mengel. I made additional small changes in terminology in the LaTeX file the tool produces: "part" is used in place of "scrap" and "definition" in place of "macro." This version, called Nuweb 0.91, is available from *http://www.cs.rpi.edu/~musser/gp/*. The only differences are in the `.tex` documentation file; the new version does not differ from previous versions in the way it produces code files from Nuweb source files.

There are a few points about Nuweb that could trip you up if you aren't aware of them:

1. Nuweb normally replaces tab characters with eight blanks. You can have it retain the tabs by putting a flag, `-t`, on the Nuweb `@o` (or `@O`) command that outputs the file. If you keep your makefiles in a Nuweb document, as in the preceding appendix, be sure to use this flag since the tabs are essential to the correct interpretation of the makefile by the `make` utility. Thus the command to output the makefile looks like this:

   ```
   @O Makefile -t
   ```

2. Another useful flag on Nuweb file output commands is `-d`. This causes `#line` directives to be inserted in the file, which are used by compilers, editors, debuggers, and other tools to relate error messages back to lines in the original source file (the Nuweb `.w` file in this case) rather than lines in the file the tool is processing. Using this feature makes it a lot easier to make corrections directly to the Nuweb source file, which is crucial to the literate programming approach—you shouldn't be even looking at, much less editing, the code files or `.tex` file produced by Nuweb.

3. That said, there seems to be a problem with the omniORB IDL compiler (`omniidl2`) getting a little confused by line directives in `.idl` files. It understands them to the extent of correctly emitting error messages in terms of the Nuweb source file, but it somehow fails to produce the correct `.h` file (containing C++ stub code). For this reason, you must *not* use the `-d` flag when outputting IDL files for input to `omniidl2`. (Or you could use the flag initially while working out any syntax errors in your IDL file, then remove it.)

   This problem is not likely to appear when using other CORBA IDL compilers, but I haven't yet tried it with any others.