complexity, though, can be largely hidden from the programmer through the use of the **Rcpp** package, and in fact the net result is that the **Rcpp** route is actually easier than using **.C()**.

Here is the **Rcpp** version of our previous code:

```
// AdjRcpp.cpp

#include <Rcpp.h>
#include <omp.h>

// the function transgraph() does the work
// arguments:
//    adjm:  the adjacency matrix (NOT assumed
//           symmetric), 1 for edge, 0 otherwise;
//            note: matrix is overwritten
//           by the function
//  return value: the converted matrix

// finds the chunk of rows this thread will process
void findmyrange(int n,int nth,int me,int *myrange)
{  int chunksize = n / nth;
   myrange[0] = me * chunksize;
   if (me < nth-1)
      myrange[1] = (me+1) * chunksize - 1;
   else myrange[1] = n - 1;
}

RcppExport SEXP transgraph(SEXP adjm)
{
   int *num1s,  // i-th element will be the number
                // of 1s in row i of adjm
       *cumul1s,  // cumulative sums in num1s
       n;
   Rcpp::NumericMatrix xadjm(adjm);
   n = xadjm.nrow();
   int n2 = n*n;
   Rcpp::NumericMatrix outm(n2,2);

   #pragma omp parallel
   {  int i,j,m;
      int me = omp_get_thread_num(),
          nth = omp_get_num_threads();
      int myrows[2];
```

```
    int tot1s;
    int outrow,num1si;
    #pragma omp single
    {
       num1s = (int *) malloc(n*sizeof(int));
       cumul1s = (int *) malloc((n+1)*sizeof(int));
    }
    findmyrange(n,nth,me,myrows);
    for (i = myrows[0]; i <= myrows[1]; i++) {
       // number of 1s found in this row
       tot1s = 0;
       for (j = 0; j < n; j++)
          if (xadjm(i,j) == 1) {
             xadjm(i,(tot1s++)) = j;
          }
       num1s[i] = tot1s;
    }
    #pragma omp barrier
    #pragma omp single
    {
       // cumul1s[i] will be tot 1s before row
       // i of xadjm
       cumul1s[0] = 0;
       // now calculate where the output of each
       // row in adjm should start in outm
       for (m = 1; m <= n; m++) {
          cumul1s[m] = cumul1s[m-1] + num1s[m-1];
       }
    }
    for (i = myrows[0]; i <= myrows[1]; i++) {
       // current row within outm
       outrow = cumul1s[i];
       num1si = num1s[i];
       for (j = 0; j < num1si; j++) {
          outm(outrow+j,0) = i + 1;
          outm(outrow+j,1) = xadjm(i,j) + 1;
       }
    }
}

Rcpp::NumericMatrix outmshort =
    outm(Rcpp::Range(0,cumul1s[n]-1),
       Rcpp::Range(0,1));
```

```
    return outmshort;
}
```

### 5.5.5 Compiling and Running

We will still run **R CMD SHLIB** to compile, but we have more libraries to specify in this case. In the **bash** shell, we can run

```
export R_LIBS_USER=/home/nm/R
export PKG_LIBS="-lgomp"
export PKG_CXXFLAGS="-fopenmp -I/home/nm/R/Rcpp/include"
```

That first command lets R know where our R packages are, in this case the **Rcpp** package. The second states we need to link in the **gomp** library, which is for OpenMP, and the third both warns the compiler to watch for OpenMP pragmas and to include the **Rcpp** header files.

Note that that last **export** assumes our source code is in C++, as indicated below by a **.cpp** suffix to the file name. Since C is a subset of C++, our code can be pure C but we are presenting it as C++.

We then run

```
R CMD SHLIB AdjRcpp.cpp
```

This produces a **.so** file or equivalent as before. Here is a sample run:

```
> library(Rcpp)  # don't forget to do this FIRST
> dyn.load("AdjRcpp.so")
> m <- matrix(sample(0:1,16,replace=T),ncol=4)
> m
      [,1] [,2] [,3] [,4]
[1,]    1    1    1    0
[2,]    1    1    0    1
[3,]    1    1    0    0
[4,]    1    0    0    1
> .Call("transgraph",m)
      [,1] [,2]
 [1,]    1    1
 [2,]    1    2
```

```
 [3,]    1    3
 [4,]    2    1
 [5,]    2    2
 [6,]    2    4
 [7,]    3    1
 [8,]    3    2
 [9,]    4    1
[10,]    4    4
```

Sure enough, we do use **.Call()** instead of **.C()**. And note that we have only one argument here, **m**, rather than five as before, and that the result is actually in the return value, rather than being in one of the arguments. In other words, even though **.Call()** is more complex than **.C()**, use of **Rcpp** makes everything much simpler than under **.C()**. In addition, **Rcpp** allows us to write our C/C++ code as if column-major order were used, consistent with R. No wonder **Rcpp** has become so popular!

### 5.5.6 Code Analysis

The heart of using **.Call()**, including via **Rcpp**, is the concept of the SEXP ("S-expression," alluding to R's roots in the s language). In R internals, a SEXP is a pointer to a C struct containing the given R object and information about the object. For instance, the internal storage for an R matrix will consist of a struct that holds the elements of the matrix and and its numbers of rows and columns. It is this encapsulation of data and metadata into a struct that enabled us to have only a single argument in the new version of **transgraph()**:

```
RcppExport SEXP transgraph(SEXP adjm)
```

The term **RcppExport** will be explained shortly. But first, note that both the input argument, **adjm**, and the return value are of type SEXP. In other words, the input is an R object and the output is an R object. In our run example above,

```
> .Call("transgraph",m)
```

the input was the R matrix **m**, and the output was another R matrix.

The machinery in **.Call()** here is set up for C, and C++ users (including us in the above example) need a line like

```
extern "C" transgraph;
```

in the C++ code. The **RcppExport** term is a convenience for the programmer, and is actually

```
#define RcppExport extern "C"
```

Now, let's see what other changes have been made. Consider these lines:

```
Rcpp::NumericMatrix xadjm(adjm);
n = xadjm.nrow();
int n2 = n*n;
Rcpp::NumericMatrix outm(n2,2);
```

**Rcpp** has its own vector and matrix types, serving as a bridge between those types in R and corresponding arrays in C/C++. The first line above creates an **Rcpp** matrix **xadjm** from our original R matrix **adjm**. (Actually, no new memory space is allocated; here **xadjm** is simply a pointer to the data portion of the struct where **adjm** is stored.) The encapsulation mentioned earlier is reflected in the fact that **Rcpp** matrices have the built-in method **nrow()**, which we use here. Then we create a new $n^2 \times 2$ **Rcpp** matrix, **outm**, which will serve as our output matrix. As before, we are allowing for the worst case, in which the input matrix consists of all 1s.

**Rcpp** really shines for matrix code. Recall the discussion at the beginning of Section 5.4.2. In our earlier versions of this adjacency matrix code, both in the standalone C and R-callable versions, we were forced to use one-dimensional subscripting in spite of working with two-dimensional arrays, e.g.

```
if (adjm[n*i+j] == 1) {
```

This was due to the fact that ordinary two-dimensional arrays in C/C++ must have their numbers of columns declared at compile time, whereas in this application such information is unknown until run time. This is not a problem with object-oriented structures, such as those in the C++ Standard Template Library (STL) and **Rcpp**.

So now with **Rcpp** we can use genuine two-dimensional indexing, albeit with parentheses instead of brackets:[4]

```
if (xadjm(i,j) == 1) {
```

Note, though, that **Rcpp** subscripts follow C/C++ style, starting at 0 rather than 1 for R. The "+1" in

[4]It is still possible to do one-dimensional indexing, using brackets, but recall that **Rcpp** uses column-major order for compatibility with R.

```
outm(outrow+j,1) = xadjm(i,j) + 1;
```

in which we were inserting a certain column number from the adjacency matrix, was needed to resolve this discrepancy.

Most of the remaining code is unchanged, except for the return value:

```
Rcpp::NumericMatrix outmshort =
    outm(Rcpp::Range(0,cumul1s[n]-1),Rcpp::Range(0,1));
return outmshort;
```

As before, we allocated space for **outm** to allow for the worst case, in which $n^2$ rows were needed. Typically, there are far fewer than $n^2$ 1s in the matrix **adjm**, so the last rows in **outm** are filled with 0s. Here we copy the nonzero rows into a new **Rcpp** matrix **outmshort**, and then return that.

All in all, **Rcpp** made our code simpler and easier to write: We have fewer arguments, arguments are in explicit R object form, we don't need to deal with row-major vs. column-major order, and our results come back in exactly the desired R object, rather than as one component of a returned R list.

### 5.5.7 Advanced Rcpp

**Rcpp** is at this writing becoming increasingly more versatile, offering several ways to set up code, other than the very basic approach presented here.

One advanced feature (many would consider it basic, not advanced) is Rcpp *attributes*, which enables simpler code, though with an extra build step. For instance, the argument **adjm** in **transgraph()** could be declared as of type **Rcpp::NumericMatrix** rather than SEXP. This would be clearer, and would save us the trouble of creating an extra variable, **xadjm**.

Another example is Rcpp *syntactic sugar*, which magically allows you to add some R-style syntax to C++, very nice!

## 5.6 Speedup in C

So, let's check whether running in C can indeed do much better than R in a parallel context, as discussed back in Section 1.1.

```
> n <- 10000
```