5.5 Example: Transforming an Adjacency Matrix, R-Callable Code

A typical application might involve an analyst writing most of his code in R, for convenience, but write the parallel part of the code in C/C++, to maximize speed. The most common interfaces for this are the R functions **.C()**, **.Call()** and **Rcpp**. We'll illustrate that notion here, modifying our earlier code for transforming an adjacency matrix, in Section 5.4.1.

5.5.1 The Code, for .C()

Code suitable for the **.C()** interface follows below.

```
1  // AdjMatXformForR.c
2
3  #include <R.h>
4  #include <omp.h>
5  #include <stdlib.h>
6
7  // transgraph() does this work
8  // arguments:
9  //    adjm:  the adjacency matrix (NOT assumed symmetric), 1 for edge, 0
10 //           otherwise; note: matrix is overwritten by the function
11 //    np:  pointer to number of rows and columns of adjm
12 //    nout:  output, number of rows in returned matrix
13 //    outm:  the converted matrix
14
15 void findmyrange(int n,int nth,int me,int *myrange)
16 {  int chunksize = n / nth;
17    myrange[0] = me * chunksize;
18    if (me < nth-1) myrange[1] = (me+1) * chunksize - 1;
19    else myrange[1] = n - 1;
20 }
21
22 void transgraph(int *adjm, int *np, int *nout, int *outm)
23 {
24    int *num1s,  // i-th element will be the number of 1s in row i of adjm
25        *cumul1s,  // cumulative sums in num1s
26        n = *np;
27    #pragma omp parallel
28    {  int i,j,m;
29       int me = omp_get_thread_num(),
30           nth = omp_get_num_threads();
31       int myrows[2];
32       int tot1s;
33       int outrow,num1si;
34       #pragma omp single
35       {
```

```
36          num1s = malloc(n*sizeof(int));
37          cumul1s = malloc((n+1)*sizeof(int));
38       }
39       findmyrange(n,nth,me,myrows);
40       for (i = myrows[0]; i <= myrows[1]; i++) {
41          tot1s = 0;  // number of 1s found in this row
42          for (j = 0; j < n; j++)
43             if (adjm[n*j+i] == 1) {
44                adjm[n*(tot1s++)+i] = j;
45             }
46          num1s[i] = tot1s;
47       }
48       #pragma omp barrier
49       #pragma omp single
50       {
51          cumul1s[0] = 0;  // cumul1s[i] will be tot 1s before row i of adjm
52          // now calculate where the output of each row in adjm
53          // should start in outm
54          for (m = 1; m <= n; m++) {
55             cumul1s[m] = cumul1s[m-1] + num1s[m-1];
56          }
57          *nout = cumul1s[n];
58       }
59       int n2 = n * n;
60       for (i = myrows[0]; i <= myrows[1]; i++) {
61          outrow = cumul1s[i];  // current row within outm
62          num1si = num1s[i];
63          for (j = 0; j < num1si; j++) {
64             outm[outrow+j] = i + 1;
65             outm[outrow+j+n2] = adjm[n*j+i] + 1;
66          }
67       }
68    }
69 }
```

We could have a **main()** function here, but instead will be calling the code from R, as will be seen shortly.

5.5.2 Compiling and Running

In writing a C file **y.c** containing a function **f()** that we'll call from R, one can compile using R from a shell command line:

```
R CMD SHLIB y.c
```

This produces a runtime-loadable library file. In Linux systems, for instance, the file **y.so** would be created, with the corresponding file for Windows being **y.dll**. We then load it from R:

```
> dyn.load("y.so")
```

after which can call **f()** from R in some manner, such as **.C()** or **.Call()**. We've written the code above to be compatible with the simpler interface, **.C()**, which takes the form

```
> .C("f",our arguments here)
```

A more complex but more powerful call form, **.Call()** is also available, to be discussed below.

The file **y.c** must include the R header file:

```
#include <R.h>
```

Generally the good thing about compiling via R CMD SHLIB is that we don't have to worry where the header file is, or worry about the library files. But things are a bit more complicated if one's code uses OpenMP, in which case we must so inform the compiler. We can do this by setting the proper environment variable. For C code and the **bash** shell, for instance, we would issue the shell command

```
% export SHLIB_OPENMP_CFLAGS = -fopenmp
```

Here is a sample run, again in the R interactive shell, with the C file being **AdjMatXformForR.c**:

```
n <- 5
dyn.load("AdjMatXformForR.so")
a <- matrix(sample(0:1,n^2,replace=T),ncol=n)
out <-.C("transgraph",as.integer(a),as.integer(n),integer(1),
    integer(2*n^2))
```

Compare this last line to the signature of **transgraph()**:

```
void transgraph(int *adjm, int *np, int *nout, int *outm)
```

Note the following:

- The return value must be of type **void**, and in fact return values are passed via the arguments, in this case **nout** (the number of rows in the output matrix) and **outm** (the output matrix itself).
- All arguments are pointers.

- Our R code must allocate space for the output arguments.

Concerning that last point, there is no longer reason to have our C code allocate memory for the output matrix, as it did in Section 5.4. Here we set up that matrix to have worst-case size before the call, as we did in the **Rdsm** version.

So, here is a test run:

```
> n <- 5
> dyn.load("AdjMatXformForR.so")
> a <- matrix(sample(0:1,n^2,replace=T),ncol=n)
> out <-.C("transgraph",as.integer(a),as.integer(n),integer(1),
+  integer(2*n^2))
> out
[[1]]
 [1] 0 0 0 1 0 1 3 0 4 1 3 4 0 0 3 4 1 0 0 4 1 1 0 1 1

[[2]]
[1] 5

[[3]]
[1] 14

[[4]]
 [1] 1 1 1 1 2 2 2 3 4 4 5 5 5 5 0 0 0 0 0 0 0 0 0 0 0 1 2 4 5 1 4 5 1 2 5 1 2 4
[39] 5 0 0 0 0 0 0 0 0 0 0 0
```

As you can see, the return value of **.C()** is an R list, with one element for each of the arguments to **transgraph()**, including the output arguments.

Note that by default, all input arguments are duplicated, so that any changes to them are visible only in the output list, not the original arguments. Here **out**[[**1**]] is different from the input matrix **a**:

```
> a
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    0    1    1
[2,]    1    0    0    1    1
[3,]    1    0    0    0    0
[4,]    0    1    0    0    1
[5,]    1    1    0    1    1
```

Duplication of the data might impose some slowdown, and can be disabled, but this usage is discouraged by the R development team.

Our output matrix, **out**[[**4**]], is hard to read in its linear form. Let's display it as a matrix, keeping in mind that our other output variable, **out**[[**3**]], tells us how many (real) rows there are in our output matrix:

```
> (nout <- out[[3]])
[1] 14
> o4 <- out[[4]]
> om <- matrix(o4, ncol=2)
> om[1:nout,]
      [,1] [,2]
 [1,]    1    1
 [2,]    1    2
 [3,]    1    4
 [4,]    1    5
 [5,]    2    1
 [6,]    2    4
 [7,]    2    5
 [8,]    3    1
 [9,]    4    2
[10,]    4    5
[11,]    5    1
[12,]    5    2
[13,]    5    4
[14,]    5    5
```

5.5.3 Analysis

So, what has changed in this version? Most of the change is due to the differences between R and C.

Most importantly, the fact that R uses column-major storage for matrices while C uses row-major order (Section 2.3) means that much of our new code must "reverse" the old code. For example, the line

```
outm[2*(outrow+j)+1] = adjm[n*i+j];
```

in the original code now becomes

```
int n2 = n * n;
```

```
...
outm[outrow+j+n2] = adjm[n*j+i] + 1;
```

5.5.4 The Code, for Rcpp

The other major way to call C/C++ code from R is via the **.Call()** function. It is considered more advanced than **.C()**, but is much more complex. That 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:

```
1  // AdjRcpp.cpp
2
3  #include <Rcpp.h>
4  #include <omp.h>
5
6  // the function transgraph() does the work
7  // arguments:
8  //    adjm:  the adjacency matrix (NOT assumed symmetric),
9  //           1 for edge, 0 otherwise; note: matrix is overwritten
10 //           by the function
11 //  return value: the converted matrix
12
13 // finds the hunk of rows this thread will process
14 void findmyrange(int n,int nth,int me,int *myrange)
15 {  int chunksize = n / nth;
16    myrange[0] = me * chunksize;
17    if (me < nth-1) myrange[1] = (me+1) * chunksize - 1;
18    else myrange[1] = n - 1;
19 }
20
21 RcppExport SEXP transgraph(SEXP adjm)
22 {
23    int *num1s,  // i-th element will be the number of 1s in row i of adjm
24        *cumul1s,  // cumulative sums in num1s
25        n;
26    Rcpp::NumericMatrix xadjm(adjm);
27    n = xadjm.nrow();
28    int n2 = n*n;
29    Rcpp::NumericMatrix outm(n2,2);
30
```

```
31    #pragma omp parallel
32    {  int i,j,m;
33       int me = omp_get_thread_num(),
34           nth = omp_get_num_threads();
35       int myrows[2];
36       int tot1s;
37       int outrow,num1si;
38       #pragma omp single
39       {
40          num1s = (int *) malloc(n*sizeof(int));
41          cumul1s = (int *) malloc((n+1)*sizeof(int));
42       }
43       findmyrange(n,nth,me,myrows);
44       for (i = myrows[0]; i <= myrows[1]; i++) {
45          tot1s = 0;  // number of 1s found in this row
46          for (j = 0; j < n; j++)
47             if (xadjm(i,j) == 1) {
48                xadjm(i,(tot1s++)) = j;
49             }
50          num1s[i] = tot1s;
51       }
52       #pragma omp barrier
53       #pragma omp single
54       {
55          cumul1s[0] = 0;  // cumul1s[i] will be tot 1s before row i of xadjn
56          // now calculate where the output of each row in xadjm
57          // should start in outm
58          for (m = 1; m <= n; m++) {
59             cumul1s[m] = cumul1s[m-1] + num1s[m-1];
60          }
61       }
62       for (i = myrows[0]; i <= myrows[1]; i++) {
63          outrow = cumul1s[i];  // current row within outm
64          num1si = num1s[i];
65          for (j = 0; j < num1si; j++) {
66             outm(outrow+j,0) = i + 1;
67             outm(outrow+j,1) = xadjm(i,j) + 1;
68          }
69       }
70    }
71
72    Rcpp::NumericMatrix outmshort =
73       outm(Rcpp::Range(0,cumul1s[n]-1),Rcpp::Range(0,1));
```

```
74      return outmshort;
75  }
```

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:[3]

```
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

[3]It is still possible to to 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.

5.6 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
> a <- matrix(sample(0:1,n^2,replace=T),ncol=n)
> system.time(out <-.C("transgraph",as.integer(a),
+    as.integer(n),integer(1),integer(2*n^2)))
   user  system elapsed
  5.692   0.852   3.193
```

Gathering our old timings, the various methods are compared in Table 5.3.

5.7 Run Time vs. Development Time

Inspecting Table 5.3, we see that going from serial R to parallel R cut down run time by about 72%, while the corresponding figure for OpenMP was 88%. To be sure, the OpenMP version was actually more than twice as