Calling C/C++ from R and Fun with OpenMP

September 24, 2019

Our focus today

What we're not doing

- developing an `R` package (see, e.g., official documentation Writing R Extensions and Hadly Wickham R packages)
- learning C/C++, FORTRAN, `R`, or OpenMP
- taking a deep look at any one topic

What we are doing

- mentioning some topics that might get you thinking about how to improve your code
- scratching the surface of some expansive topics in computing
- providing some code and ideas that might point you in the right direction

Why connect `R` and lower-level code

`R` is an interpreted language and, as a result, can be slow at

- vectorizing loops whose subsequent iterations depend on previous iterations
- executing recursive functions

Also, we often want to use data structures, algorithms, and libraries written in lower-level code (e.g., BLAS, LAPACK, CHOLMOD, Eigen, GNU Scientific Library, etc.).

We could just write standalone code, but

- `R` is nice for input/output and other tasks in-between
- we might want to use some of `R`'s C functions, e.g., RNGs and distributions in `Rmath.h`
- we might eventually write an `R` package
- simplifies in-house code sharing and teaching

R Foreign Language Interfaces

The authoritative document is “Writing R Extensions” found at https://cran.r-project.org.

We’ll focus on calling C/C++ for now (but calling FORTRAN and JAVA is similar). There are three approaches for passing stuff between R and C/C++.

1. `.Call()` designed for calling code that understands R objects and environments. Allows multiple arguments to be passed to C/C++ and R objects returned.
2. `External()` like `.Call()` but the C/C++ function is passed as a single argument containing a LISTSXP, a pairlist from which the arguments can be extracted.
3. `.C()` (and `.Fortran`) designed to call code that does not know about R. Straightforward, but limited types of arguments and all checking of arguments must be done in R. No return value, but may alter its arguments.

Why not Rcpp?

There are some real advantages to using Rcpp

- Rcpp API (Application Programming Interface) "protects you from many of the historical idiosyncrasies of the R API"–Hadley Wickham
- takes care of memory management
- provides helper methods to working with R objects in C++
- many more advantages, see, e.g., http://www.rcpp.org/

Sounds good, so why use R's API?

- preference to write standalone flexible C/C++ code, then with slight modification can be called from R
- one fewer level of abstraction to deal with (and perhaps some overhead)—feels closer to the metal
- it's what I know well and how most R packages with source code are written

Moving between R and C/C++ types

Mapping between the modes of R atomic vectors and the types of arguments to a C function or FORTRAN subroutine.

R storage mode	C type	FORTRAN type
logical	int *	INTEGER
integer	int *	INTEGER
double	double *	DOUBLE PRECISION
complex	Rcomplex *	DOUBLE COMPLEX
character	char **	CHARACTER*255
raw	unsigned char *	none

Getting started

Calling a C/C++ function from R requires two pieces: a C/C++ function and an R wrapper function that uses `.Call()`.

Compile the C/C++ code and call from R as a shared object `.so` (Linux or MacOS X) or as a `.dll` (Windows).

From within R load the compiled object using `dyn.load()` and unload it using `dyn.unload()`.

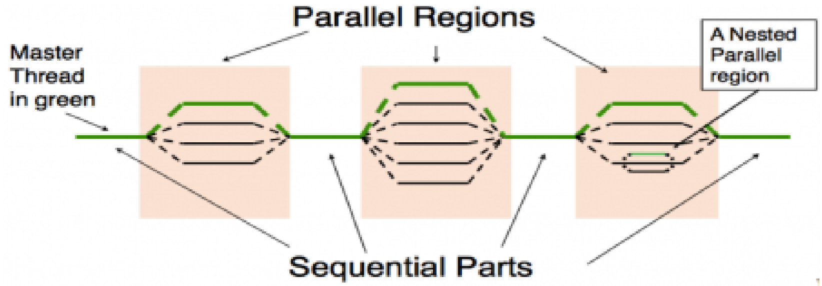
Exercise #2

Let's turn to Exercise 2 `cIDist.R` and `cIDist.cpp` files.

Parallel computing with `OpenMP`

`OpenMP` is an industry standard API of C/C++ and FORTRAN for shared memory parallel programming.

`OpenMP` is based on two concepts: the use of threads (think CPUs) and the fork/join model of parallelism:



All threads have access to the same shared global memory. Each thread has access to its private variables and common variables[1].

[1]figure credit: www.nersc.gov

Parallel computing with `OpenMP`

Some advantages

- high-level directives (`pragma`) used to define parallel regions simplify coding and decisions
- parallelism can be added incrementally
- compilers (or you) can optimize the number of threads needed by parallel region

"With great power comes great responsibility"—Benjamin Parker (a.k.a Uncle Ben)

You must be sure that what you are doing in parallel regions is **thread-safe**—it's very easy to make mistakes that compile without error.

Exercise #2 revisited

Consider Exercise 2 but now `cIDistOMP.R` and `cIDistOMP.cpp` files.

Code gone wrong (then right)

Let's construct a spatial correlation matrix using the Matern function such that the i, j-th element is equal to

$$R(\theta)_{i,j} = \frac{1}{2^{\nu-1}\Gamma(\nu)}(d_{i,j}\phi)^{\nu}\mathcal{K}_{\nu}(d_{i,j}\phi); \phi > 0, \nu\, 0, \qquad (1)$$

where $\theta = (\phi, \nu)$ with ϕ controlling the decay and ν controlling smoothness, Γ is the Gamma function, and $\mathcal{K}_{\nu}$ is a modified Bessel function of the second kind with order ν.

In `R` speak this is

```
(D*phi)^nu/(2^(nu-1)*gamma(nu))*besselK(x=D*phi, nu=nu)
```

and C using `Rmath.h` functions

```
pow(D[i]*phi, nu)/(pow(2, nu-1)*gammafn(nu)*
                   bessel_k(D[i]*phi, nu, 1.0)
```

Exercise #3

Consider Exercise 3 but now `cRMaternOMPWrong.cpp` and `cRMaternOMPWrong.cpp` files.

Now making it thread-safe

The problem is that `R`'s `bessel_k` C function is not thread-safe. Note the `bk` vector is allocated within `bessel_k.c`.

Instead use undocumented `bessel_k_ex` in `bessel_k.c` as illustrated in `cRMaternOMPSafe.cpp`.

Here we:

- allocate enough working space for each thread outside the parallel region
- use each thread's id (i.e., $0, 1, \ldots, \text{nTheads} - 1$) via `omp_get_thread_num()` to index the working space passed to `bessel_k_ex`