R news and tutorials contributed by (573) R bloggers

- Home
- About
- RSS
- add your blog!
- R jobs
- Contact us

Welcome!

Follow

Here you will find daily **news and tutorials about R**, contributed by over 573 bloggers. There are many ways to **follow us -**

By e-mail:

Your e-mail here

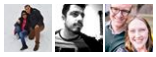
18204 readers
BY FEEDBURNER

On Facebook:

R blog...
26 391 Me gu...

Me gusta esta página

Sé el primero de tus amigos en indicar que te gusta esto.



If you are an R blogger yourself you are invited to add your own R content feed to this site (**Non-English** R bloggers should add themselves- here)

Jobs for R-users

- Product Analyst (@ Pittsburgh)
- Quantitative Developer/Data Scientist (@ Mountain View)
- Data Scientist AWF
- Network Link analysis Consultant (@ London)
- Seeking R

Programmers for long term contract role

Search & Hit Enter

Popular Searches

- heatmap
- web scraping
- maps
- undefined
- hadoop
- shiny
- twitter
- boxplot
- animation
- ggplot2
- trading
- finance
- time series
- LaTeX
- excel
- ggplot
- pca
- quantmod
- googlevis
- eclipse
- market research
- RStudio
- how to import image file to R
- tutorial
- knitr
- rattle
- coplot
- rcmdr
- gis
- sweave

Recent Posts

- Differences in the network structure of CRAN and BioConductor
- SWMPr 2.1.0 on CRAN
- Visualizing bdsns data using bdvis
- How Do Auction Values Differ by the Number of Teams in Your League? A Multilevel Model
- How do you know if your data has signal?
- Multilevel Models and Political Advertising
- JSM 2015 [day #2]

- The Keep Function
- On radical manuscript openness
- How Do You Know if Your Data Has Signal?
- In case you missed it: July 2015 roundup
- Share your Shiny Apps with Docker and Kitematic!
- JSM 2015 [day #1]
- Curl 0.9.2: tweaks and proxies for windows
- Turning your R (or Python) models into APIs

Other sites

- SAS blogs
- Jobs for R-users
- Statistics of Israel

OpenMP Tutorial, with R Interface

January 17, 2015
By matloff

(This article was first published on **Mad (Data) Scientist**, and kindly contributed to R-bloggers)

Almost any PC today is multicore. Dual-core is standard, quad-core is easily attainable for the home, and larger systems, say 16-core, are easily within reach of even smaller research projects. In addition, large multicore systems can be "rented" on Amazon EC2 and so on.

The most popular way to program on multicore machines is to use OpenMP, a C/C++ (and FORTRAN) callable system that runs on Linux, Mac and Windows. (For Macs, you need the OpenMP-enabled version of Mac's **clang** compiler.)

This blog post will **present a short tutorial on OpenMP**, including calling OpenMP code from R, using **Rcpp**. Use of the latter will be kept to basics, so if you are also new to **Rcpp**, you'll learn how to use that too. This tutorial is adapted from my book, *Parallel Computation for Data Science: with Examples in R, C/C++ and CUDA,* to be published in June 2015.

I'll assume that you know R well, and have some familiarity with C/C++.

Threaded programming:

Most programs running on multicore systems are *threaded*. This means that several invocations, called threads, of the given program are running simultaneously, typically one thread per core. A key point is that the threads share memory, making it easy for them to cooperate. The work of the program is divided among the threads, and due to the simultaneity, we potentially can achieve a big speedup.

Writing threaded code directly is messy, so higher-level systems have been developed to hide the messy details from the programmer, thus making his/her life far easier. As noted, OpenMP is the most popular of these systems, though Intel's Threading Building Blocks system is also widely used.

Our example:

Here we convert a graph adjacency matrix to a 2-column matrix that lists the edges. (You don't need know what graphs are, which will be explained shortly.) Here is an example of running our code, a function **transgraph()**:

```
> m
     [,1] [,2] [,3] [,4]
[1,]    0    1    1    0
[2,]    1    0    1    1
[3,]    1    1    1    0
[4,]    0    1    1    0
> .Call("transgraph",m)
      [,1] [,2]
 [1,]    1    2
 [2,]    1    3
 [3,]    2    1
 [4,]    2    3
 [5,]    2    4
 [6,]    3    1
 [7,]    3    2
 [8,]    3    3
 [9,]    4    2
[10,]    4    3
```

Here our matrix **m** represented a 4-vertex, i.e. 4-node graph. If you are not familiar with graphs, think of set of 4 Web sites. Row 1 of **m** says that site 1 has links to sites 2 and 3, site 2 has links to sites 1, 3 and 4 and so on. The output matrix says the same thing; e.g. the (1,2) in the first row says there is a link from site 1 to site 2, and so on.

We might have thousands of Web sites, or even more, in our analysis. The above matrix conversion thus could be quite computationally time-consuming, hence a desire to do it in parallel on a multicore system, which we will do here.

Plan of attack:

The idea is simple, conceptually. Say we have four cores. Then we partition the rows of the matrix **m** into four groups. Say **m** has 1000 rows. Then we set up four threads, assigning thread 1 to work with rows 1-250, thread 2 dealing with rows 251-500, etc. Referring to the eventual output matrix (the 2-column matrix seen above) as **mout**, each thread will compute its portion of that matrix, based on that thread's assigned rows.

There is one tricky part, which is to stitch the outputs of the four threads together into the single matrix **mout**. The problem is that we don't know ahead of time where in **mout** each thread's output should be placed. In order to do that, our code will wait until all the threads are done with their assigned work, then ask how many output rows each one found.

To see how this helps, consider again the little four-site example above, and suppose we have just two threads. Thread 1 would handle rows 1-2 of **m**, finding four rows of **mout**, beginning with (1,2) above. These will eventually be the first four rows of **mout**. The significance of that is that the output of thread 2 must start at row 5 of **mout**. In this manner, we'll know where in **mout** to place each thread's output.

The code:

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

```
// 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;
}

// SEXP is the internal data type for R
// objects, as seen from C/C++;
// here our input is an R matrix adjm, and
// the return value is another R matrix
RcppExport SEXP transgraph(SEXP adjm)
{
   // i-th element of num1s will be the
   // number of 1s in row i of adjm
   int *num1s,
       *cumul1s,  // cumulative sums in num1s
       n;
   // make a C/C++ compatible view of the
   // R matrix;
   // note: referencing is done with
   // ( , ) not [ , ], and indices start at 0
   Rcpp::NumericMatrix xadjm(adjm);
   n = xadjm.nrow();
   int n2 = n*n;
   // create output matrix
   Rcpp::NumericMatrix outm(n2,2);

   // start the threads; they will run the
   // code below simultaneously, though not
   // necessarily executing the same line
   // at the same time
   #pragma omp parallel
   {  int i,j,m;
      // each thread has an ID (starting at 0),
      // so determine the ID for this thread
      int me = omp_get_thread_num(),
          // find total number of threads
          nth = omp_get_num_threads();
      int myrows[2];
      int tot1s;
      int outrow,num1si;
      // have just one thread execute the
      // following block, while the
      // others wait
      #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;
      }
      // wait for all threads to be done
      #pragma omp barrier
      // again, one thread does the
      // following, others wait
      #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 xadjm should start
         // in outm
         for (m = 1; m <= n; m++) {
            cumul1s[m] =
               cumul1s[m-1] + num1s[m-1];
```

```
         }
      }
      // now this thread will put the rows it
      // found earlier into outm
      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;
         }
      }
   }
   // have some all-0 rows at end of outm;
   // delete them
   Rcpp::NumericMatrix outmshort =
      outm(Rcpp::Range(0,cumul1s[n]-1),
      Rcpp::Range(0,1));
   return outmshort;  // R object returned!
}
```

(For your convenience, I've place the code here.)

Compiling and running:

Here's how to compile on Linux or a Mac. In my case, I had various files in **/home/nm**, which you'll need to adjust for your machine. From the shell command line, do

```
export R_LIBS_USER=/home/nm/R
export PKG_LIBS="-lgomp"
export PKG_CXXFLAGS="-fopenmp -I/home/nm/R/Rcpp/include"
R CMD SHLIB AdjRcpp.cpp
```

This produces a file **AdjRcpp.so**, which contains the R-loadable function, **transgraph()**. We saw earlier how to call the code from R. However, there is one important step not seen above: Setting the number of threads.

There are several ways to do this, but the most common is to set an environment variable in the shell *before* you start R. For example, to specify running 4 threads, type

```
export OMP_NUM_THREADS=2
```

Brief performance comparison:

I ran this on a quad core machine, on a 10000-row graph **m**. The pure-R version of the code (not shown here) required 27.516 seconds to run, while the C/C++ version took only 3.193 seconds!

Note that part of this speedup was due to running four threads in parallel, but we also greatly benefited by running in C/C++ instead of R.

Analysis of the code:

I've tried to write the comments to tell most of the details of the story, but a key issue in reading the code is what I call its "anthropomorphic" view: We write the code pretending we are a thread!

For example, consider the lines

```
   #pragma omp parallel
   {  int i,j,m;
      // each thread has an ID (starting at 0),
      // so determine the ID for this thread
      int me = omp_get_thread_num(),
          // find total number of threads
          nth = omp_get_num_threads();
      int myrows[2];
      ...  // lots of lines here
      for (i = myrows[0]; i <= myrows[1];
            i++) {
         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;
         }
      }
   }
```

As explained in the comments: The pragma tells the compiler to generate machine code that starts the threads. Each of the threads — say we have four — will simultaneously execute all the lines following the pragma, through the line with the closing brace. However, each thread has a different ID number, which we can sense via the call to **omp_get_thread_num()**, which we record in the variable **me**. We write the code pretending we are thread number **me**, deciding which rows of **adjm** we must handle, given our ID number.

Debugging:

I'm a fanatic on debugging tools, and whenever I see a new language, programming environment etc., the first question I ask is, How to debug code on this thing?

Any modern C/C++ debugging tool allows for threaded code, so that for example the user can move from one thread to another while single-stepping through the code. But how does one do this with C/C++ code that is called from R? The answer is, oddly enough, that we debug R itself!

But though the R Core Team would greatly appreciate your help in debugging R 🙂 we don't actually do that. What we do is start R with the **-d** option, which places us in the debugger. We then set a breakpoint in our buggy code, e.g. for **gdb**

```
(gdb) break transgraph
```

We then resume execution, placing us back in R's interactive mode, then call our buggy function from there as usual, which places in that function — and in the debugger! We can then single-step etc. from there.

Other OpenMP constructs:

The above example was designed to illustrate several of the most common pragmas in OpenMP. But there are a lot more tricks you can do with it, and in addition there are settings that you can make to improve performance. Some of these are explained in my book, with more examples and so on, and in addition there are myriad OpenMP tutorials (though not R-oriented) on the Web.

Happy OpenMP-ing!

Comments: 12

Related

Parallel Programming with GPUs and R
by Norman Matloff
We've heard that graphics processing units — GPUs — can bring big increases in
In "R bloggers"

Using OpenMP-ized C code with R
What is OpenMP? Basically a standard compiler extension allowing one to easily distribute calculations
Similar post

Comments: 13
Rdsm: a Flexible Parallel Computation Package for R
In "R bloggers"

To **leave a comment** for the author, please follow the link and comment on his blog: **Mad (Data) Scientist**.

R-bloggers.com offers **daily e-mail updates** about R news and tutorials on topics such as: visualization (ggplot2, Boxplots, maps, animation), programming (RStudio, Sweave, LaTeX, SQL, Eclipse, git, hadoop, Web Scraping) statistics (regression, PCA, time series, trading) and more...

Comments are closed.

Top 3 Posts from the past 2 days

- Scatterplots
- In-depth introduction to machine learning in 15 hours of expert videos
- Installing R packages

Top 9 articles of the week

1. In-depth introduction to machine learning in 15 hours of expert videos
2. Installing R packages
3. Scatterplots
4. Using apply, sapply, lapply in R
5. Turning your R (or Python) models into APIs
6. Basics of Histograms
7. Microsoft Launches Its First Free Online R Course on edX
8. Read Excel files from R
9. Adding a legend to a plot

Sponsors