parallel computing in R

With computational requirements and datasets growing as they have in the present, **parallel operation** in R has be developed amongst several available tools. In R, failed **parallel processing** mechanisms are sometimes discovered when certain programs actually run faster without **parallel processing**.

Given the example of a network graph (weblinks and social networks, etc.), **A** represents the ***adjacency matrix*** of the graph (e.g. **A[3,8]** is **1** or **0**, dependent on whether there is a link from **node 3** to **node 8**). For a given two vertices (two websites), the **manual outlinks** (outbound links common to two websites) become the problem's focus. The mean number of manual outlinks, averaged over all pairs of websites in the data is computed using the below for a **n-by-n** $\mathbb{R}^{n \times n}$ matrix:

```
sum = 0
for i = 0...n-1
  for j = i+1...n-1
    for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
mean = sum / (n*(n-1)/2)
```

The above code could be computationally expensive given the potential size of the dataset. The latter context is often dealt with by **chunking** the data into separate parts and processing in **multiple locations**.

The **snow** (Simple Network of Workstations) package in R is a simple and convenient form of **parallel processing** in R. The code for the manual outlinks problem above is presented as follows with **snow**:

```
# snow version of mutual links problem

mtl <- function(ichunk,m) {
  n <- ncol(m)
  matches <- 0
  for (i in ichunk) {
    if (i < n) {
      rowi <- m[i,]
      matches <- matches +
        sum(m[(i+1):n,] %*% rowi)
      }
   }
  matches
}

mutlinks <- function(cls,m) {
  n <- nrow(m)
  nc <- length(cls)
  # determine which worker gets which chunk of i
  options(warn=-1)
  ichunks <- split(1:n,1:nc)
  options(warn=0)
  counts <- clusterApply(cls,ichunks,mtl,m)
  do.call(sum,counts) / (n*(n-1)/2)
}
```

The code from the illustration above in applying the **snow** package runs as follows:

- … load code
- … load the **snow** library
- … form a **snow** cluster
- … set up ***adjacency matrix*** of focus
- … run code on the latter matrix on the formed **snow** cluster

Given the above code stored in ***SnowMutLinks.R***, the following commands execute for dual-core machines:

```
> source("SnowMutLinks.R")
> library(snow)
> cl <- makeCluster(type="SOCK",c("localhost","localhost"))
> testm <- matrix(sample(0:1,16,replace=T),nrow=4)
> mutlinks(cl,testm)
[1] 0.6666667
```

The above code starts two new R processes locally. Given the two new processes and the existing process where the code was previously entered, there are three total R processes. The origin R process forms a ***cluster*** in **snow** parlance (named **cl** above). The **snow** package uses parallel processing in the context of a ***scatter/gather*** paradigm:

- … The origin process partitions the data into ***chunks*** and sends them to the newly created parallel process (***scatter phase***)
- … The newly created processes process their received ***chunks***
- … The origin process collects the results from the created processes (***gather phase***) and combines them appropriately

The communication between the processes is established through ***network sockets*** (discussed previously):

```
> testm
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    1
[2,]    0    0    0    0
[3,]    1    0    1    1
[4,]    0    1    0    1
```

The **test matrix** above shows row **1** has zero common outlinks with row **2**; row **1** has two outlinks in common with row **3**; Row **1** has one outlink in common with Row **4**; Row **2** has zero outlinks in common with any; Row **3** has one outlink in common with Row **4**. There are four total manual outlinks out of $4 \times \frac{3}{2} = 6$ pairs—the mean value of $\frac{4}{6} = 0.6666667$ as shown in the previous output.

Clusters can be made of any size depending on the available resources. Given **3 dual-core** machines named ***p28***, ***p29***, ***p30***, the following creates a ***six-process cluster*** using the **snow** package.

```
> cl6 <- makeCluster(type="SOCK",c("pc28","pc28","pc29","pc29","pc30","pc30"))
```

analyzing the snow code in R

To examine the **mutlinks( )** function, the number of **rows in the matrix** and the **number of processes** (aside from the origin process) within the **cluster** (line 17 and 18 above):

The created process is then determined as to which value of **i** in the **for i** loop will be handled. The **split( )** function is the appropriate use for the latter task; in a 4-row matric and a 2-process cluster:

```
> split(1:4,1:2)
$`1`
[1] 1 3

$`2`
[1] 2 4
```

The above set up has one R process work on the odd values of **i** and the other work on the even values of **i**. Potential warnings from the **split( )** function ("data length is not a multiple of split variable") are avoided through the **options( )** function.

The heavy lifting in the **mutlinks( )** function is performed where the **snow clusterApply( )** function is called. The latter calls the same specified function (**mtl( )** in this case) with specific arguments for each process and optional arguments that apply to all processes. The following is performed in line 23 of **mutlinks( )**:

- … New Process 1 directed to call the **mtl( )** function with arguments **ichunks[[1]]** and **m**
- … New Process 2 calls the **mtl( )** function with arguments **ichunks[[2]]** and **m**; and so one for all processes
- … Each process performs their task and returns results to the Origin Process
- … The Origin Process collects all results into an R list, assigned above to **counts**

All of the elements of the **count** variable are summed:

```
1   > sum(1:2,c(4,10))
2   [1] 17
```

Note the above call only applies to numeric vectors, therefore the **do.call( )** function is used for the **R list**. The vectors are extracted from **counts** and then passed through the **sum( )** function appropriately. Note the vectorized implementation in lines 9 and 10; opposed to using **for j** and **for k loops**.

obtainable speedup ⏫ in R

In many parallel-processing applications, ***overhead*** exists (wasted time on noncomputational activity). There exists overhead in the form of the time needed to send the matrix from the origin program to the newly created processed. Additional overhead is encountered when sending the function **mtl( )** to the workers. Lastly, after the created processes complete their roles, overhead exists in the return to the origin process.

resorting to C ⚙ in R

Another option is to write R code that parallels C/C++. In the context of multi-core machines, parallel computation is performed with ***threads***; analogous to the created processes through the **snow** package. The latter is constructed as a ***shared-memory*** system. All cores access the same RAM, changes made across threads are visible to the other threads.

The below example applies the **mutual outlinks** problem in R-callable code with the **OpenMP** package in R:

```
1   #include <omp.h>
2   #include <R.h>
3
4   int tot; // grand total of matches, over all threads
5
6   // processes row pairs (i,i+1), (i,i+2), ...
7   int procpairs(int i, int *m, int n)
8   { int j,k,sum=0;
9     for (j = i+1; j < n; j++) {
10      for (k = 0; k < n; k++)
11        // find m[i][k]*m[j][k] but remember R uses col-major order
12        sum += m[n*k+i] * m[n*k+j];
13    }
14    return sum;
15  }
16  void mutlinks(int *m, int *n, double *mlmean)
17  { int nval = *n;
18    tot = 0;
```

(continued on the proceeding page)

```
19    #pragma omp parallel
20    { int i,mysum=0,
21      me = omp_get_thread_num(),
22      nth = omp_get_num_threads();
23      // in checking all (i,j) pairs, partition the work according to i;
24      // this thread me will handle all i that equal me mod nth
25      for (i = me; i < nval; i += nth) {
26        mysum += procpairs(i,m,nval);
27      }
28      #pragma omp atomic
29      tot += mysum;
30    }
31    int divisor = nval * (nval-1) / 2;
32    *mlmean = ((float) tot)/divisor;
33  }
```

The **OpenMC** library is linked through the **–fopenmp** and **–lgomp** option below (source file ***romp.c***):

```
1   gcc -std=gnu99 -fopenmp -I/usr/share/R/include -fpic -g -O2 -c romp.c -o romp.o
2   gcc -std=gnu99 -shared -o romp.so romp.o -L/usr/lib/R/lib -lR -lgomp
```

The code is tested in R as follows:

```
1   > dyn.load("romp.so")
2   > Sys.setenv(OMP_NUM_THREADS=4)
3   > n <- 1000
4   > m <- matrix(sample(0:1,n^2,replace=T),nrow=n)
5   > system.time(z <- .C("mutlinks",as.integer(m),as.integer(n),result=double(1)))
6     user system elapsed
7     0.830 0.000 0.218
8
9   > z$result
10  [1] 249.9471
```

Threads can be specified in **Open MP** through the operating system variable **OMP_NUM_THREADS**. Operating system variables can be set in R with the **Sys.setenv( )** function (set to 4 threads above). Additionally, R has other ***byte-compilation*** functions such as **cmpfun( )**.

The **OpenMC** package code is compiled in C, with the addition of ***pragmas***, instructing the compiler to insert library code to perform **OpenMP** operations (threads activated in line 21). Note the importance of variable scope in the above code. The **mysum** variable in line 21 is for each thread to maintain its own sum. Line 4 contains the global variable **tot** which is shared amongst all threads. Each thread contributes to the total on line 30.

The **nval** variable on line 18 is also a global variable (although a local variable in terms of C) to all threads. An additional ***pragma*** (**atomic**) is located on line 29 and is applied to line 30 (opposed to the entire block). The **atomic *pragma*** serves to avoid a ***race condition***; where two threads are updating a variable synonymously.

The origin thread executes in lines 18, 19, and within the **.C( )** function in R that calls the **mutlinks( )** function. The origin thread becomes idle in line 21 when the new processes are activated; until they finish in line 31 (the origin process then reactivates).

The **procpairs()** function accesses matrix **m** as a one-dimensional array (common in C; recalling the difference of reading matrices column by column in R and row-wise in C) indexed at 0 (R indexes at 1).

The result is the multiplication on line 12. The factors are the **(k,i)** and **(k,j)** elements of **m** in the C code, which are factors **(i+1,k+1)** and **(j+1,k+1)** elements in the R code.

other OpenMP programs in R

A ***barrier*** in parallel processing refers to the line of code where threads interact:

```
1   #pragma omp barrier
```

Execution is suspended upon reaching a ***barrier*** until all threads catch up (useful in iterative algorithms). The above is an ***explicit barrier***, while other pragmas have ***implicit barriers*** within the code blocks (***single*** and ***parallel***; an implied barrier is located on line 31, causing the origin process to remain dormant).

A block that follows the ***omp critical pragma*** is considered a ***critical section***, where threads are executed individually. The ***omp critical pragma*** functions as the ***atomic pragma*** discussed above with the exception of the latter being limited to a single statement:

```
1   #pragma omp critical
2   {
3     // place one or more statements here
4   }
```

The block of code following an ***omp single pragma*** is executed by a single thread:

```
1   #pragma omp single
2   {
3     // place one or more statements here
4   }
```

The above is useful for initializing sum variables amongst threads; preventing simultaneous thread activity.

gpu programming in R

An additional shared-memory parallel hardware includes ***graphics processing units*** (**GPU**). The brief context in application is to write the R code interfaced to parallel C; opposed to writing parallel R code. **GPU**s follow the ***threads*** model at a greater magnitude (as much as 100 cores). Consequentially, multiple threads are capable of running within the same block. Access to **GPU**s begin on the **CPU** serving as the ***host***. The code is executed on the **GPU** (***device***) by transferring the data to the ***device***.

general performance considerations in R

It is important to understand the context of ***overhead*** inducing aspects of the code being executed in R. An example is the used of **shared-memory machines** as illustrated previously. Because the latter process requires waiting on each thread at a time, ***overhead*** inevitably occurs. Additionally, cores might contain ***caches*** that require ***overhead*** to keep the ***caches*** consistent amongst cores. **GPU**s can suffer from ***latency***, as the delay in time when the initial information bits are received from memory. ***Overhead*** also exists between transfers from the ***host*** and ***devices***.

To illustrate the topic of **static** and **dynamic task assignment**, the following code is from **OpenMP** above:

```
26  for (i = me; i < nval; i += nth) {
27    mysum += procpairs(i,m,nval);
28  }
```

The variable **me** represents the thread number, keeping threads from overlapping on values of variable **i**. The preassignment of tasks for each thread is what is referred to as **static task assignment**.

An alternative approach to the above illustration is presented on the proceeding page (alliteration intended) by revising the original **for loop**:

```
int nexti = 0; // global variable
...
for ( ; myi < n; ) { // revised "for" loop
  #pragma omp critical
  {
    nexti += 1;
    myi = nexti;
}
if (myi < n) {
  mysum += procpairs(myi,m,nval);
  ...
  }
}
...
```

The above approach is what is referred to as a **dynamic task assignment**, where the values of **i** which are handled by the threads is not predetermined as before. It can be assumed that using the **dynamic task assigning** approach will lead to efficiency gains. An example would be if the completion times of predetermined threads do not finish as expected, causing a ***load balance*** problem. Conversely, the above could cause ***caching*** issues that render **dynamic task assignment** much slower in certain context.

Conclusively, it is appropriate to use **static task assignments** in most circumstances cited below:

> The standard deviation of the sum of independent, identically distributed random variables, divided by the mean of that sum, goes to zero as the number of terms goes to infinity. In other words, sums are approximately constant. This has a direct implication for load-balancing concerns: Since the total work time for a thread in static assignment is the sum of its individual task times, that total work time will be approximately constant; there will be very little variation from thread to thread, [finishing together]

Another issue applies to the following algorithm from the **mutual outlinks** illustration previously discussed:

```
sum = 0
for i = 0...n-1
  for j = i+1...n-1
    for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
mean = sum / (n*(n-1)/2)
```

Given that **n = 10000** with **4-threads**, there are multiple approaches to partitioning the **for i loop**. A massive inefficient would result from assigning the first thread to have the **i** values from **0-2499** and the second thread to handle **2500-4999**, etc; the result being a large ***load imbalance*** (a thread processing a given value of **i** performs a proportional amount of work to **n-i**). The latter is reasoning behind the staggered assignment of threads in the **mutual outlinks** example. **Static assignment** might ultimately require more preparation and analysis, but can be avoided through best practices or methods such as random assignment to threads.

debugging parllel code ↔ in R

Some parallel R packages, such as **Rmpi**, **snow**, **foreach**, etc. (at the publishing of Norman Matloff's book), do not create terminal windows for individual processes; removing the option to leverage R's debugger. An available solution would require debugging an underlying function; by setting up artificial values and applying R's ordinary debugging mechanisms. Assuming a bug exists in an actual argument itself, the process becomes more difficult considering trace information is not readily returned as the **print( )** and **message( )** functions might not apply to all parallel R packages. The **cat( )** function should remain available for these particular circumstances in using parallel R packages; writing the contents to an actual file.