Parallel package

Steven Chiou

This document gives a brief introduction to one of R's most useful standard packages – the `parallel` package.

**Motivating example

Lets start with a motivating example. Suppose we want to find the number of primes between **2 and 5,000,000** (inclusive). A sensible approach would consists of the following steps:

1. Write a function to detect if a given integer is a prime.
2. Apply the function in #1 on all integer between 2 and 5,000,000.
3. The total number of primes between 2 and 5,000,000 is the number of primes detected in step #2.

Here is a function that can be used to check if a given integer is a prime:

```
> is.prime <- function(n) ifelse(n == 2 || sum(!(n %% 1:ceiling(sqrt(n)))) == 1, TRUE, FALSE)
> is.prime(2) ## 2 is a prime
[1] TRUE
> is.prime(197) ## 197 is a prime
[1] TRUE
> is.prime(19791) ## 19791 is not a prime
[1] FALSE
```

The following code gives the number of prime numbers between 2 and 5,000,000 while avoiding the use of `for` loops.

```
> system.time(print(sum(sapply(2:5e6, is.prime))))
[1] 348513
   user  system elapsed
 45.836   0.132  45.973
```

**Parallel computing

By default, `R` handles loops sequentially. What it means is, `R` will compute the first task at hand and moves on to the next when the first is complete.

$$\boxed{\text{Task 1}} \rightarrow \boxed{\text{Task 2}} \rightarrow \boxed{\text{Task 3}} \rightarrow \cdots \rightarrow \boxed{\text{Task 10000}}$$

There are `R` packages (e.g., `snow`, `multicore`, `parallel`, `Segue`, etc.) allow `R` to take advantage of multiple processes to compute tasks in a parallel fashion

$$\begin{cases} \boxed{\text{Task 1}} \rightarrow \boxed{\text{Task 5}} \rightarrow \boxed{\text{Task 10}} \cdots \\ \boxed{\text{Task 2}} \rightarrow \boxed{\text{Task 7}} \rightarrow \boxed{\text{Task 8}} \cdots \\ \boxed{\text{Task 3}} \rightarrow \boxed{\text{Task 9}} \rightarrow \boxed{\text{Task 11}} \cdots \\ \boxed{\text{Task 4}} \rightarrow \boxed{\text{Task 6}} \rightarrow \boxed{\text{Task 16}} \cdots \end{cases}$$

Of these packages in parallel computing, only the parallel package is a standard package, meaning you can load the package without having to install it.

```
> library(parallel)
```

The first step is to know how many CPU cores (clusters or workers) are available. This gives us an idea on how to divide the jobs

```
> detectCores()
```

```
[1] 8
```

This means I can assign jobs to up to 8 clusters. To assign jobs, the following steps are needed: 1. Define clusters 2. Setup clusters 3. Export functions/libraries to each cluster 4. Assign jobs to each cluster 5. Shut down clusters The following codes give an example for the above prime example:

```
> cl <- makePSOCKcluster(detectCores())
> setDefaultCluster(cl)
> clusterExport(NULL, "is.prime")
> system.time(print(sum(parSapply(NULL, 2:5e6, FUN = is.prime))))
```

```
[1] 348513
```

```
   user  system elapsed
  2.892   0.248  15.544
```

```
> stopCluster(cl)
```

Within the `parallel` environment, `sapply` is replaced with `parSapply`. The `parallel` also offers functions like `parLapply` and `parApply` to replace with `lapply` and `apply`.

Note that the computing time with `parSapply` in this example is not proportional to the computing time without. The lost in computing time is due to the communication between processes. A (possibly) more efficient way for our example is to submit a batch of jobs to each cluster. For example:

```
> cl <- makePSOCKcluster(detectCores())
> setDefaultCluster(cl)
> clusterExport(NULL, "is.prime")
> system.time(print(sum(parApply(NULL, matrix(1:5e6, 5e4), 1, FUN = function(x) sapply(x, is.prime))) - 1))
```

```
[1] 348513
```

```
   user  system elapsed
  0.160   0.016  11.878
```

```
> stopCluster(cl)
```

The first example assigns 5,000,000 jobs to 8 clusters while the second example assigns 50,000 jobs (each job has 100 numbers to compute) to 8 clusters.