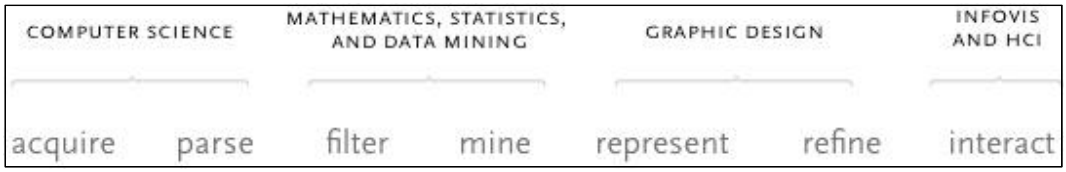
R로 하는 멀티코어 병렬 프로그래밍

전희원 : 약 7년의 검색관련 개발경력을 가지고 있으며, 기계학습 검색 랭킹, Spelling Suggestion같은 대용량 로그 기반 지능형 서비스 애플리케이션 개발을 해왔다. 현재 NexR 에서 Data Scientist로 재직하고 있으며 freesearch.pe.kr 블로그를 운영하고 있다.

1회 연재에서 R에 대한 개략적인 문법이나 사용방법에 대해서 설명하는 시간을 가졌다. 지난번 연재에서 R의 언어적인 특징을 살펴보았으며 충분히 익숙하게 다룰 수 있는 개인적인 시간을 보냈다면 이번 회를 이해하기가 훨씬 수월할거라 생각한다.
R은 리눅스의 철학과 굉장히 유사한 특징을 가지고 있다. S+라는 상용툴에서 사용하는 S언어의 GNU 버전이며, S+가 Unix라고 한다면 R은 리눅스와 상당히 유사한 성격을 가지고 태어났다고 볼 수 있다. 토발즈가 리눅스 커널을 만들었고, 나머지 대부분 시스템 관련 명령어나 기능들은 사용자의 기여(contribution)으로 현재까지 유지되고 있다. R도 비슷하다, 뉴질랜드 오클랜드 대학의 로스 이하카와 로버트 젠틀맨에 의해서 R언어가 최초로 개발되었으며(현재는 R코어팀에 의해서 릴리즈 되지만) 언어 이외의 부분들에 대해서는 대부분 패키지 형태로 사용자들의 기여에 의해서 발전이 되고 있다. 이 패키지들의 개발 속도는 2000년도 초반부터 2010년초까지 조사에 의하면 각 버전이 릴리즈 될 때마다 지수승으로 패키지의 숫자가 증가하고 있다는 조사[1)]가 있으며 최신의 데이터 마이닝 알고리즘부터 가장 오래된 알고리즘까지 모두 패키지화 되어 개발되고 있는 상황이다. 많은 패키지의 존재 자체가 R언어가 데이터 분석에서 프로토타이핑(prototyping)이 가장 빠른 툴로서 존재하게 하지 않나 하는 생각을 해본다. 이번 연재와 다음 연재 이 두 연재는 바로 이런 패키지들에 대한 설명이 주를 이루게 될 것이다.
빅 데이터 분석 붐으로 인해 병렬처리, 분산처리 관련해서도 굉장히 활발하게 개발되고 있으며 패키지도 상당히 다양하니 한번 눈여겨 볼 만 할 거라 생각한다.

왜 R 데이터 분석에서 병렬 처리가 필요한가?
R언어 자체로는 병렬 처리를 지원하지 않는다. 이는 C, C++, Python과 같은 언어들이 언어 레벨에서 지원하지 않고 라이브러리 형식으로 지원하는 것과 상당히 유사하나 R이라는 언어가 병렬 처리를 지원하지 않는다고 한다면 뭔가 새로울 것만 같은 R이라는 언어에 대해서 실망을 하게 될지도 모를 거라는 생각이 먼저든다. 사실 필자가 생각하기에 언어 레벨에서 지원하는 언어는 Erlang과 같은 함수형 언어정도가 언어 레벨에서 병렬화를 지원하는 정도가 되지 않을까 한다.
따라서 R스크립트를 실행시키면 멀티코어 CPU에서 단일 코어만 사용하게 된다. 멀티코어에서 단일 코어만 사용한다는 것은 어찌 보면 상당히 아까운 자원을 놀리게 되는 것이다.
그럼 데이터 분석이라는 작업의 어떤 부분에서 병렬 처리를 하는게 좋은가?
이 이야기를 하기 전에 데이터 과학(data science) 관련 업무들이 어떤 것들로 구성되어 있는지 확인해볼 필요가 있다.

1) http://blog.revolutionanalytics.com/2010/01/r-package-growth.html



위 그림은 벤자민 프라이(Benjamin Fry)[2]의 글에서 가져온 그림으로 데이터 분석, 마이닝의 제반 과정을 언급하고 있다. 데이터를 가져와서 데이터 클리닝 작업과 같은 파싱, 간단한 전처리 작업을 하게 된다. 데이터가 많을 경우 이 부분은 병렬, 분산 처리가 필수적인 부분이다. 따라서 컴퓨터 사이언스 영역의 것이라고 구분을 한다. 두 번째 부분은 방대한 데이터에서 어떤 문제를 해결하기 위한 필요 데이터를 선별하고 이들을 적절하게 전처리 하여 통계 모델링을을 하거나 EDA(Exploratory Data Analysis)를 하여 새로운 비즈니스 이슈들을 발견하게 되는 과정이다. 만일 여기서 어떤 분류 모델을 만든다고 한다면 수많은 테스트를 수행하게 되고 테스트 결과에 따라 다시 데이터 분석 작업을 하는 과정을 수도 없이 반복하게 된다. 바로 이 반복되는 과정 자체가 병렬화의 이유가 된다. 그리고 나머지 과정은 마이닝 결과에 대한 효과적인 시각화의 이슈를 보여주고 있다. 사족이지만 데이터 과학자(Data Scientist)는 컴퓨터 사이언스 베이스에 통계나 데이터 마이닝 지식이 아주 풍부하고 그래프 디자인 실력과 더불어 효과적인 시각화 감각이 있는 사람이여야 된다는 것을 이 그림이 보여주고 있다. 한 개만 잘 하기도 힘든 이 시대에 참 많은 것들을 요구하는 직무가 아닐 수 없다. 일단 최근 빅 데이터 관련 컨퍼런스에서는 첫 번째 과정에 대해서 집중해서 조명하고 있으며 그 중심에 하둡(Hadoop)이라는 플랫폼이 있으나 아직 두 번째 과정에 대한 이슈는 아직 그리 조명 받지 못하고 있다.

필자는 예전 프로젝트에서 모델링을 수행할 시 약 500여개의 모델을 만들었다. 추천 모델 빌드하는데만 개당 3시간 정도 소요되었다. 모두 테스트 모델이였고 이 500개중에 가장 성능이 좋은 1개만이 선택 되었다. 이 시간은 25일동안 모델만 빌드하는데 들었다는 것을 의미한다. 물론 중간중간 여러 가설과 속성들에 대한 분석 작업이 있기는 했지만 매번 빌드시 이 시간을 단축하면 얼마나 좋을까 수도 없이 생각했었다. 필자가 이야기 하고 싶은 튜닝 포인트는 바로 이런 부분이다. 독립적 수행이 가능한 프로세스들을 병렬로 수행하는 작업은 데이터 마이닝 모델링 분야에서 어찌보면 가장 필요한 부분이 아닐까 한다.

그 일환으로 이번 회에서는 멀티코어 리소스를 충분히 활용할 수 있는 방법을 소개하도록 하겠다.

R 병렬 처리 라이브러리들

R 병렬 처리 라이브러리들은 HPC(High Performance Computing)이라는 카테고리로 CRAN에 분류되어 있다. 따라서 아래 링크를 따라가면 HPC에 관련된 모든 패키지들을 확인 가능하다.

http://cran.r-project.org/web/views/HighPerformanceComputing.html

여기서 명시적(Explicit) 병렬과 내재적(Implicit) 병렬 라이브러리로 나눠 있는데, 이 차이는 사용자가 명시적으로 병렬화 설정을 필요로 할 경우에 명시적 병렬 라이브러리로 분류되고 별다른 설정없이(코드의 변경 없이) 병렬화를 사용 가능하다면 내재적 병렬로 분류된다.

2) http://benfry.com/phd/dissertation-050312b-acrobat.pdf

명시적 병렬 라이브러리 : Rmpi, snowfall

내재적 병렬 라이브러리 : doSMP

doSMP

doSMP는 Revolution Analytics라는 회사에서 배포하는 패키지이다. 이 회사는 Revolution R이라는 R 사용 버전을 판매하는 회사로서 최근 R이 약한 빅 데이터 분석관련 기능을 보강하고 doSMP같은 멀티코어에 최적화된 라이브러리를 배포하고 있기도 하다. 이 패키지를 설치하려면 아래와 같은 명령어를 사용하면 된다.

```
> install.packages("doSMP")
```

이 패키지를 설치하면 foreach라는 패키지도 자동으로 설치가 되는데, doSMP가 하는 역할은 이 foreach의 멀티코어 병렬 처리의 백엔드를 담당한다고 생각하면 된다. foreach문에서 %dopar%라는 연산자를 붙여주게 되면 동일한 코드로 멀티코어 병렬 처리가 가능한 코드가 되는 것이다. 하지만 이 doSMP는 싱글 머신에서만 동작한다. 그러니까 클러스터 기반으로는 동작하지 않는다는 것이다. 이는 이 패키지를 싱글머신 싱글코어의 컴퓨팅 환경에서 사용해야 별 효과가 없을 거라는 것을 의미한다. 만일 클러스터 기반에서 멀티코어까지 모두 활용한 컴퓨팅을 하고자 한다면 나중에 설명할 Rmpi나 snowfall과 같은 패키지를 사용하길 권한다.

하지만 그런 단점에서도 불구하고 많은 분석가들의 가려운 곳을 긁어주는 패키지 중에 하나다. 그러나 대부분의 분석가들은 컴퓨터 전문가가 아니여서 클러스터 환경에서 R을 돌릴 생각을 잘 하지 못하며 더군다나 리눅스 머신 설정을 하길 기대할 만한 사람들이 아니다. 그런 의미에서 연산자 하나만으로 멀티코어 병렬 처리를 지원하는 이 패키지야 말로 가장 적합한 패키지가 아닐까 한다. 일단 부트스트래핑(bootstrapping) 예제를 하나 보도록 하자.

부트스트래핑은 데이터를 학습셋과 테스트셋으로 쪼개서 여러번 테스트 하는 방법으로 생각하면 된다. 데이터 셋 자체가 독립적으로 존재 가능하니 모델 빌드시 병렬로 처리할 수 있는 부분이다.

```
> library(doSMP)
> w <- startWorkers(workerCount =2)
> # doSMP에 worker를 등록한다.
> registerDoSMP(w)
> #binary classification 문제로 바꾸기 위해 setosa 클래스를 제거했으며, 첫 번째 속성과 마지막 종속변수만 남기고 x에 데이터를 할당한다.
> x <- iris[which(iris[, 5] != "setosa"), c(1, 5)]
> #10000개의 모델 생성 예정
> trials <- 10000
> #5000개의 청크로 나뉘게 될 것이다.
> chunkSize <- ceiling(trials/getDoParWorkers())
> smpopts <- list(chunkSize = chunkSize)
> # r이라는 변수에 모든 coefficients 결과가 matrix자료구조로 누적된다.
> ptime <- system.time({
+ r <- foreach(icount(trials),.combine = cbind, .options.smp = smpopts) %dopar%
+ {
```

```
+   ind <- sample(100, 100, replace = TRUE)
+   result1 <- glm(x[ind, 2] ~ x[ind, 1], family = binomial(logit))
+   coefficients(result1)
+ }
+ })[3]
> ptime
elapsed
 90.42
```

<코드.1 doSMP를 이용한 멀티코어 병렬 버전>

```
> ptime <- system.time({
+ r <- foreach(icount(trials),.combine = cbind, .options.smp = smpopts) %do%
+ {
+   ind <- sample(100, 100, replace = TRUE)
+   result1 <- glm(x[ind, 2] ~ x[ind, 1], family = binomial(logit))
+   coefficients(result1)
+ }
+ })[3]
> ptime
elapsed
 142.12
```

<코드.2 순차 실행 버전>

위 코드는 1회 연재에서 사용했던 iris 데이터를 기반으로 logistic regression을 랜덤 샘플링 해서 10000개의 모델을 만들어 보는 코드이다. 물론 이 코드의 경우 테스트셋을 나누고 테스트셋에 대한 테스트 결과를 누적시켜야 하지만 목적상 그런 내용을 생략한 코드이다.

필자의 PC는 윈도우7 32bit 운영체제이며 4G 메모리에 코어 두 개가 있는 CPU를 가지고 있다. workerCount를 2로 둔 이유가 코어 두 개인 이유와 같다고 생각하면 된다. %do% 연산자를 사용하게 되면 순차적으로 코드를 처리하게 되는데 이렇게 해서 걸린 시간은 총 140.12초였다. 이를 멀티코어 병렬로 처리하게 하기 위해서 %do%를 %dopar%로 바꾸어 실행시켜 보니 90.42초가 걸린다는 것을 알 수 있다. 이 차이는 좀더 복잡한 모델과 데이터 구조를 사용할 때 그 격차는 더 커지게 된다.

doSMP는 멀티코어 병렬 처리를 병렬 처리에 대한 최소한의 지식으로 수행 가능하게 해주는 고마운 패키지라고 말할 수 있다. 게다가 Window나 Mac OS X, Linux 등 대부분의 운영체제를 지원하고 있어서 아주 편리하게 사용 가능하다. 더 자세한 세부 기능들은 "http://cran.r-project.org/web/packages/doSMP/index.html" 페이지를 참고하기 바란다.

Rmpi

Rmpi에 대한 설명에 들어가기 앞서 MPI(Message Passing Interface)에 대한 설명을 하도록 하겠다. MPI는 프로세스가 병렬로 돌기 위한 환경을 의미하며 프로세스간의 커뮤니케이션이 메시지 전달을 통해 이루어진다. MPI자체가는 분산 및 병렬처리의 기술의 기술 표준만을 제안하고 구현 방법이라든지 구체적인 프로토콜은 기술하고 있지 않다. 그냥 기술 표준 스펙이라고 인식하면 된다.

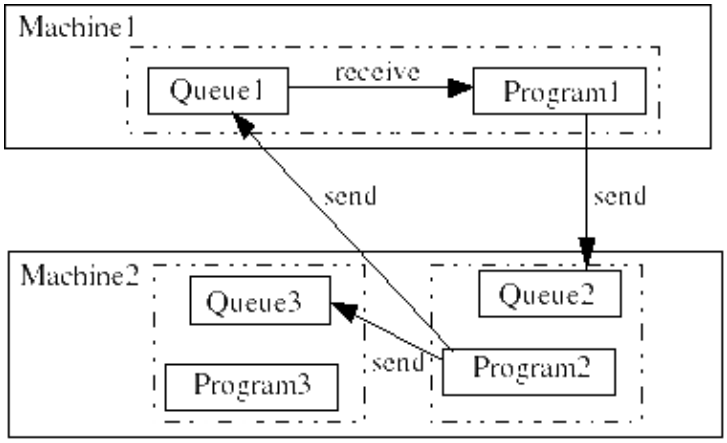


그림 1 MPI 예제

실제로 메시지 전달 방식은 위 그림1을 통해 쉽게 이해할 수 있다.

각 MPI 프로그램은 메일박스(queue)를 가지고 있으며 어떤 MPI 프로그램도 다른 MPI 프로그램의 메일박스에 메시지를 넣을 수 있다. 그렇게 받은 메시지들은 박스에 쌓이고 각 프로그램은 프로세싱이 끝날 때마다 메일박스를 확인해 새로운 메시지를 처리하게 된다. 그리고 처리된 메시지들은 메시지를 보낸 프로그램에 전달되는 과정을 거치게 된다.

그림1을 통해 설명하면 프로그램2가 프로그램1과 프로그램3에 메시지를 전달했으며 프로그램1에서 처리된 메시지가 다시 프로그램2로 보내진 것을 알 수 있다.

예를 들면 처리해야 되는 프로세싱 단위를 쪼개서 각 분산된 슬레이브(slave) MPI 프로그램에 던지고 처리된 메시지를 마스터에서 최종적으로 받으면 되는 간단한 구조의 병렬처리 방식이다.

아래는 MPI의 대표적인 구현체 들이다.

MPICH and MPICH2
LAM-MPI
OpenMPI

일단 필자의 리눅스 OS 싱글 머신에서 멀티코어 병렬을 돌려보는 예제를 시도해 보도록 하자. 윈도우 머신에서는 MPICH2 정도를 사용 할 수 있는데, 사실 리눅스 머신에서 셋팅하는 것보다 더 복잡하다. 따라서 리눅스 머신으로 해보도록 하겠다.

일단 필요한 라이브러리들을 설치한다. 물론 root로 작업하면 된다.

```
yum install openmpi openmpi-devel openmpi-libs
ldconfig /usr/lib/openmpi/1.4-gcc/ #라이브러리 설치 경로
```

~/.bash_profile에 아래와 같은 라이브러리 서치 패스를 넣어준다(경로는 독자 환경에 맞게 설정해 주시면 된다).

```
export
LD_LIBRARY_PATH="${LD_LIBRARY_PATH}${LD_LIBRARY_PATH:+:}/usr/lib/openm
pi/1.4-gcc/"
```

R 콘솔을 실행 시킨 후 아래 명령어로 Rmpi를 설치하면 완료 된다. 물론 include path와 library path는 독자분들의 환경에 맞게 설정하면 된다.

```
install.packages("Rmpi", configure.args
=c("--with-Rmpi-include=/usr/lib/openmpi/1.4-gcc/include/",
"--with-Rmpi-libpath=/usr/lib/openmpi/1.4-gcc/lib/",
"--with-Rmpi-type=OPENMPI"))
```

테스트를 위해서 아래와 같은 명령어를 입력해 본다.

코드 설명을 간단히 하자면 먼저 Rmpi 라이브러리를 로딩하고 슬레이브 2개를 만든다. 로그를 보자면 슬레이브 2개에 마스터 하나가 생성되었다는 것을 볼 수 있다. 그리고 각 슬레이브에 자신의 랭크를 리턴하게 명령어를 내린 결과이다.

```
> library("Rmpi")
> mpi.spawn.Rslaves(nslaves=2)
        2 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 3 is running on: i-10-26-6-38
slave1 (rank 1, comm 1) of size 3 is running on: i-10-26-6-38
slave2 (rank 2, comm 1) of size 3 is running on: i-10-26-6-38
> mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",mpi.comm.size()))
$slave1
[1] "I am 1 of 3"

$slave2
[1] "I am 2 of 3"

> mpi.close.Rslaves()
[1] 1
> mpi.quit()
```

대부분의 Rmpi 프로그램의 구조는 다음과 같다.

1. 먼저 Rmpi 라이브러리를 로딩하고 슬레이브들을 생성한다.
2. 데이터 처리에 필요한 함수 제작
3. 함수를 워커 함수화 하기 위해 코드 래핑을 함
4. 데이터 초기화
5. 필요한 모든 데이터와 함수들을 슬레이브에 전달한다.
6. 슬레이브가 워커 함수를 실행하게끔 함
7. **슬레이브와 커뮤니케이션하면서 컴퓨테이션 결과를 확인한다.**
8. 각 결과를 리스트 형태로 받아서 처리한다.
9. 슬레이브들을 해제하고 mpi를 종료한다.

7번이 병렬화 되어 실행되는 부분이다.

아래 코드는 피보나치 수열에 대한 MPI 코드이다.

```
library(Rmpi)
mpi.spawn.Rslaves(nslaves=2)
#피보나치 수열에 대한 결과를 리턴하는 함수 물론 슬레이브들에게 전송된다.
fib <- function(n)
{
    if (n < 1)
        stop("Input must be an integer >= 1")
    if (n == 1 | n == 2)
        1
    else
        fib(n-1) + fib(n-2)
}

#모든 슬레이브가 전송받을 워커 함수
slavefunction <- function() {
    # 보내는 메시지에 대한 프로토콜 :
    #      1=테스크 처리 준비 완료, 2=테스크 완료, 3=종료
    # 받는 메시지에 대한 프로토콜 :
    #      1=테스크, 2=테스크 완료
    junk <- 0

    done <- 0
    while (done != 1) {
        # 테스크 처리 준비 완료 메시지 전송
        mpi.send.Robj(junk,0,1)

        # 테스크 수신
        task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag())
        task_info <- mpi.get.sourcetag()
        tag <- task_info[2]

        if (tag == 1) {
            # 테스크 수행 및 처리 결과 전송

            #테스크: 피보나치 수열 처리
            n <- task$n
            val <- ifelse(n == 1 | n == 2, 1, fib(n-1) + fib(n-2))
            print(val)
            res <- list(result=val, n=n)
```

```
            # 테스크 결과 전송
            mpi.send.Robj(res,0,2)
            }
        else if (tag == 2) {
            # 마스터가 모든 테스크가 종료 되었다고 알려옴
            done <- 1
        }
        # 여타 다른 메시지들은 무시하거나 에러 로그 출력
    }

    # 마스터에게 슬레이브가 종료한다고 알려줌
    mpi.send.Robj(junk,0,3)
}

#슬레이브에게 슬레이브 함수와 피보나치 함수를 전송함
mpi.bcast.Robj2slave(slavefunction)
mpi.bcast.Robj2slave(fib)
#모든 슬레이브에게 슬레이브 함수를 실행하게 함
mpi.bcast.cmd(slavefunction())

# 테스크의 list를 생성함
tasks <- vector('list')
for (i in 1:25)
{
        tasks[[i]] <- list(n=i)
}



junk <- 0
closed_slaves <- 0
n_slaves <- mpi.comm.size()-1

results <- rep(0,25)

#슬레이브의 테스크가 모두 종료 될 때까지 반복
while (closed_slaves < n_slaves) {
    # 슬레이브로부터 메시지 수신
    message <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag())
    message_info <- mpi.get.sourcetag()
```

```
    slave_id <- message_info[1]
    tag      <- message_info[2]

    if (tag == 1) {
        # 슬레이브 테스크 처리 준비 완료. 다음 테스크 전달 혹은
        # 테스크가 없을 경우 완료 태그 전송
        if (length(tasks) > 0) {
            # Send a task, and then remove it from the task list
            mpi.send.Robj(tasks[[1]], slave_id, 1);
            tasks[[1]] <- NULL
            }
        else {
            mpi.send.Robj(junk, slave_id, 2)
            }
        }
    else if (tag == 2) {
                # 테스크 처리 결과가 포함된 메시지임
                # 메시지 결과를 result 벡터에 저장함
                res <- message$result
                results[message$n] <- res
        }
    else if (tag == 3) {
        # 슬레이브 프로세스 파괴되었다는 정보 전송 받음
        closed_slaves <- closed_slaves + 1
        }
    }

# 전송된 결과로 요망하는 다름 프로세싱을 처리
# ...

# 슬레이브들을 클로즈 하고 나감..
mpi.close.Rslaves()
mpi.quit()
```

위 코드는 Rmpi 튜토리얼[3]에 있는 코드에서 피보나치 수열 함수를 래핑한 예제이다. 상세한 내용들은 코드의 주석을 확인하면 그 흐름을 알 수 있을 것인데, 결과적으로 위에서 언급한 전형적인 Rmpi 프로그램의 구조와 동일하다는 것을 알 수 있을 것이다. 사실 이 피보나치 결과는 로컬 머신에서 싱글 코어로 동작한 결과보다 느리다. 왜냐면 슬레이브로 오브젝트들을 전송하고 전송 받는 커뮤니케이션 과정 자체가 피보나치 수열 계산을 하는 리소스보다 더 소요 되기 때문이다.

3) http://math.acadiau.ca/ACMMaC/Rmpi/

따라서 이 예제에서 보는 것처럼 모든 문제들이 병렬화 해서 더 빨리지지는 않는다는 것을 알 수 있다. 그러나 fib 함수보다 더 복잡도가 높은 처리를 하는 코드라면 이 코드의 가치는 길이만큼이나 빛을 발휘할 수 있을 거라 생각한다.

Rmpi는 MPI에 익숙하다면 크게 어렵지 않게 멀티코어 뿐만 아니라 클러스터 기반의 분산 프로세싱이 가능한 환경을 쉽게 만들어 준다. 하지만 멀티코어 병렬처리 정도라면 doMPI가 훨씬 더 편한 환경을 제공하고 있음을 코드량을 보고도 알 수 있을거라 생각한다. 다음에 소개할 snowfall은 MPI를 래핑해서 더 편한 병렬 처리 프레임웍을 제공하고 있으니 snowfall을 좀 더 살펴보도록 하자.

snowfall

일단 snowfall 패키지를 설명하기 전에 snowfall의 기반이 되는 snow(Simple Network of Workstations)에 대한 설명부터 해야 할 듯 하다.

snow는 MPI, NWS(NetWork Space via new), PVM(Parallel Virtual Machine), Sockets(OS)를 기반으로 멀티코어 병렬 뿐만 아니라 클러스터 기반의 분산처리에 대한 인터페이스를 제공하고 있다. 이런 snow의 모든 인터페이스 기능과 더불어 snowfall은 필요한 경우에 순차 처리를 제공하고 있으며, 별도의 계산 함수, 에러 핸들링 등 여타 다른 병렬 처리를 편하게 해주는 함수들을 제공하고 있다. 게다가 클러스터를 관리하는 sfCluster와 결합하여 편리한 클러스터 기반 분산처리를 할 수 있게 해준다. snow에 별로 익숙지 않은 유저들에게는 아주 편리한 패키지가 snowfall이다.

snowfall이 분산을 위해 래핑을 하는 함수군이 apply군의 함수들이다. 이는 데이터를 쪼개서 특정 함수를 적용해서 결과를 리턴하는 함수군 들인데, 병렬처리 하기 좋은 성격의 함수들이다. 따라서 이 함수들에 대한 설명을 하는 시간을 가져보겠다.

```
> x <- 1:20
> dim(x) <- c(5,4)
> x
     [,1] [,2] [,3] [,4]
[1
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
> apply(X=x, MARGIN=1, FUN=max)
[1] 16 17 18 19 20
```

일단 apply함수의 경우 array자료구조나 matrix자료구조의 행이나 열에 FUN에 해당하는 함수를 적용해 vector를 리턴하게 된다.

같은 계열의 함수인 lapply는 vector나 list의 각 요소에 함수를 적용하는 하여 list를 리턴하게 하는 함수이다.

```
> x <- as.list(1:5)
> lapply(x, function(x) 2^x)
[[1]]
[1] 2
```

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

[[3]]
[1] 8

[[4]]
[1] 16

[[5]]
[1] 32
```

sapply는 lapply의 경우 list를 리턴하는것과는 다르게 vector, matrix, array를 리턴한다. mapply는 sapply의 다변수 버전이라고 생각하면 된다.

snowfall에서는 sfLapply, sfSapply, sfApply라는 함수들을 제공하는데, 이들은 각각 lapply, sapply 그리고 apply의 병렬버전이라고 생각하면 된다.

일반적으로 snowfall의 경우 아래와 같은 프로그램 구조를 가지게 된다.

1. sfInit()으로 초기화, 필요할 경우 클러스터를 셋업하고 내부 함수들을 설정한다. snowfall의 함수들을 사용하기 전에 반드시 먼저 호출해 줘야 한다.
2. 필요한 변수나 객체들을 슬레이브들에 전송한다.
3. snowfall의 계산 함수들을 활용해 병렬 기반의 계산을 한다. 이 과정은 사용자가 원하는 만큼 반복될 수 있다.
4. sfStop() 호출을 통해 병렬 실행을 종료한다.

일단 snowfall을 설치하기 위해 “install.packages(”snowfall“)”명령으로 설치하자.

snowfall은 싱글 랩탑에서 멀티코어 병렬을 처리하기 위해 소켓을 사용한다. 따라서 sfInit을 호출시 type을 명시하지 않으면 소켓기반의 별렬 처리를 수행하게 된다. 이번 원고의 목적이 멀티코어 병렬 처리니 이 snowfall만으로 충분하다. 만일 클러스터 기반의 병렬 처리를 하고자 한다면 MPI나 LAM-MPI 관련 라이브러리가 모든 클러스터 머신에 설치되어 있어야 하고 SSH기반의 passwordless 로그인 설정을 모두 해야 한다. 이 부분에 대해서는 관련 튜토리얼을 참고하기 바란다.

앞에서 Rmpi를 활용한 피보나치 수열 함수의 병렬 버전을 snowfall을 기반으로 작성해 보면 아래와 같다.

```
> sfInit( parallel=TRUE, cpus=2)
snowfall 1.84 initialized (using snow 0.3-5): parallel execution on 2 CPUs.
#fib 함수는 이전에 작성한걸 사용한다.
> sfExport("fib")
> sfLapply(1:25, fib)
[[1]]
[1] 1
```

```
[[2]]
[1] 1

[[3]]
[1] 2

[[4]]
[1] 3
.....

[[25]]
[1] 75025

> sfStop()

Stopping cluster
```

매우 간단한 코드로 바뀌었다. sfExport("fib")는 fib함수객체를 모든 슬레이브들에 전송하는 역할을 한다. 역시 이 부분도 Rmpi에 비해서 잘 인터페이싱 된 듯하다. 그리고 바로 sfLapply(1:25, fib)를 호출해 결과물을 list객체로 받게 된다.

snowfall에 대한 개략적인 소개를 마무리 짓고자 한다. snowfall의 경우 방대한 기능 덕분에 이정도의 튜토리얼은 소개 정도에 그치지 않는다. 만일 더 관심이 있다면 http://cran.r-project.org/web/packages/snowfall/index.html 이 링크를 확인해 학습해 보길 추천한다. snowfall도 여타 다른 HPC 패키지와 비슷하게 그다지 참고할만한 리소스가 부족하다. 그 시작점으로 필자의 소개가 도움이 되었길 바랄 뿐이다.

Wrap Up!

doSMP, Rmpi, snowfall에 대해서 간단히 살펴보는 시간을 가졌다. 개인적으로 멀티코어만 활용하는 로컬 머신에서의 컴퓨팅은 doSMP를 사용하는걸 추천하고 클러스터 확장 가능한 코드에서는 snowfall사용을 권한다. Rmpi는 아무래도 snowfall과 같은 상위 라이브러리가 사용하는 하위 라이브러리의 성격이 강해져 버린게 아닐까 하는 생각이 들어 사용 추천은 좀 꺼려지기도 하지만 개념상 어떻게 동작하는지는 간단하게 코드로 설명했다.

단일 R 콘솔에서 메모리 한계 이하의 데이터를 가지고 10-fold-cross validation과 같은 데이터 마이닝 관련 테스트 연산을 한다면 위와 같은 병렬처리 라이브러리들을 반드시 사용하길 권한다. 코어가 많을수록 수행 시간은 더 줄어들게 되며 그만큼 테스트 속도가 높아져 빨리 원하는 모델을 찾을 수 있기 때문이다.

R을 사용할 때 snowfall 클러스터까지 활용해야 될 경우는 극히 드물지 않을까 한다. R의 경우 메모리 한계 문제가 있으며 데이터가 크지 않다면 아무리 복잡도가 높은 연산이라도 수십대정도의 클러스터를 사용할 경우는 거의 없을 듯 하다. 하지만 빅 데이터와 복잡도가 큰 연산의 경우 메모리 한계 문제와 분산처리를 어떻게 할 것인가의 문제에 반드시 봉착하게 될 것이다. 이에 대한 내용은 다음회에 집중해서 풀어보도록 하겠다.

참고문헌

http://www.bytemining.com/2010/07/taking-r-to-the-limit-part-i-parallelization-in-r/

http://cran.r-project.org/web/packages/doSMP/vignettes/gettingstartedSMP.pdf