parallel 软件包

R 核心开发者

翻译：*yixuan* (http://yixuan.cos.name/cn/)

From 统计之都 (http://cos.name/)

最近更新：*2013* 年 *6* 月 *15* 日

1 简介

parallel 软件包最早是在 R 2.14.0 被引入的，它基于 CRAN 软件包 **multicore** (Urbanek, 2009–present) 和 **snow** (Tierney *et al.*, 2003–present) 已有的成果加以构建，并提供了两个软件包大部分的功能，同时还统一了生成随机数的操作。

在计算过程中并行可以在多个层次上加以实现：本软件包基本上是“粗粒度并行”。在最精细的粒度上，现代 CPU 可以将若干个基本操作同时进行（例如整数和浮点数的代数运算）。此外，一些外部的 BLAS 库可以利用多线程来对部分的向量/矩阵操作进行并行化。一些 R 软件包则利用 OpenMP 或 pthreads 库来进行 C 语言层面的并行。

本软件包考虑的是更多数据块下的并行计算。一个典型的例子是将一个相同的 R 函数应用到许多不同的数据集上，例如 bootstrap 计算中模拟出的多组数据集（或随机数序列本身就是要操作的数据）。这一问题的关键在于，数据块之间是彼此不相关的，并且不需要互相通信。通常，这些数据块上的计算会消耗近似相同的时间。对于此类问题，一个基本的计算模型为

1. 开启 M 个“工人”进程，并且对其进行初始化；
2. 将任务所需的数据发送给工人；
3. 将任务划分为大致相等的 M 份，然后将数据块（包括所需的 R 代码）发送给工人；
4. 等待所有工人完成各自的任务，并请求它们的结果；
5. 重复步骤 2 到 4，完成后续所有的任务；
6. 关闭工人进程。

其中的初始化过程可能包括加载软件包，以及初始化随机数序列等。

本软件包中的 `mclapply` 和 `parLapply` 函数是对这一模型的具体实现，它们几乎可以无缝地替代 `lapply`。

另一个略有不同的模型是将任务分为 $M_1 > M$ 个数据块，然后将前 M 个数据块分配给工人，等待任意一个工人完成任务，然后将剩下的某一个任务分配给它。请参阅“负载均衡”一节。

原则上，工人可以通过线程[1]或轻量级的进程来实现，但在当前的实现方式中，每一个工人都是完整的进程。它们可以通过如下三种方式进行创建：

[1]之所以是“原则上”，是因为 R 的解释器并不是线程安全的。

1. 通过 `system("Rscript")` 或类似的方法在当前机器（或一台类似的具有相同 R 版本的机器）上创建一个新的进程。接下来需要一种主进程和工人进程之间进行通信的机制，这通常通过 socket 来实现。

   这种方式在所有的 R 平台上都可以使用，尽管可以预见的是，操作系统主动的安全防御措施可能会阻止进程间的 socket 通信。Windows 和 Mac OS X 的用户可能会看到防火墙弹出的对话框，询问 R 进程是否应该接受连接。

   我们采用 `snow` 中的术语，将那些利用 socket 来监听主进程命令的工人进程称为节点“集群”。

2. 通过分流 (forking)。*分流 (Fork)* 是 POSIX 操作系统中的概念[2]，它在除 Windows 之外的所有 R 平台都能得到实现。分流将创建一个新的 R 进程，并将主进程进行一次完整的复制，包括主线程的工作空间和随机数序列的状态。然而，这种复制会（在适当的操作系统中）共享主进程的内存，直到内存发生改变。因此，分流通常是非常快的。

   分流方法最早是被 **multicore** 软件包采用的。

   注意到由于分流会复制整个进程，它将同样共享所有的 GUI 组件，例如 R 控制台以及屏幕上的图形设备。这可能会造成严重的问题。[3]

   主进程与工人进程之间需要有一种通信机制。同样地，这里有若干种实现方式，因为主进程与工人进程是共享内存的。在 **multicore** 中，原始的进程会将一个 R 表达式发送给工人进程，然后主进程打开一个管道，用于读取工人返回的结果。**parallel** 包支持这种方式，同时还支持通过 socket 进行通信的方式。

3. 利用操作系统级别的设施来将任务发送给群组内的其他机器。这种方式有若干种实现的办法，例如 **snow** 软件包可以通过 **Rmpi** 包来使用 MPI（“消息传递接口”）。在这种方式下，通信开销 (overheads) 可能会占据大部分的计算时间，所以这种方法一般用于高速连接在一起的计算机群。

   使用这一方式的 CRAN 软件包包括 **GridR**（利用 Condor 或 Globus）和 **Rsge**（利用 SGE，现在称为“Oracle Grid Engine”）。

   parallel 中提供的类似于 **snow** 的函数都可以接受 **snow** 集群作为参数，其中包括 MPI 集群。然而在本手册的后续部分，我们并不考虑这种可能。

并行计算的格局随着共享内存式多核计算机的到来也发生了改变。直到 2000 年代末，进行并行计算的集群还主要是单核或双核的计算机。然而到了现在，即使是笔记本电脑都配备了双核或四核，而具有 8 个或更多核的服务器更是非常常见的配置。**parallel** 软件包在设计上就是想利用这样的硬件设施。当然，它也能用于通过（快速的）以太网连接在一起的计算机群，甚至每台计算机可以运行不同的操作系统，而只要求它们运行了相同版本的 R。

注意到以上所有的这些通信方法都使用了 `serialize/unserialize` 函数来在进程间发送 R 对象，而对象序列化之后的长度有所限制（通常为几亿个元素），因此一个进行了良好设计的算法应该避免达到这个限制。

2 CPU/核的数量

在进行并行计算时，确定可用的 CPU 或核的数量往往是非常有帮助的。然而，这是一个相对不确定的概念。当前绝大多数的物理 CPU 都包含了两个或多个核，它们大体上是独立运行的（它们将共享

[2] http://en.wikipedia.org/wiki/Fork_(operating_system)

[3] 在 Mac OS X 下需要格外小心，例如子进程会继承 `R.app` 和 `quartz` 设备的事件循环。这些信息可以通过 C 级别的 `Rboolean` 变量 `R_isForkedChild` 进行获取。

内存，并可能共享部分的缓存）。然而，在某些处理器上这些核本身还可以同时运行多重的任务，并且在某些操作系统（如 Windows）中，我们有*逻辑* CPU 的概念，其数量可能会超过实际的核的个数。

需要注意的是，程序所能决定的是可用的 CPU 和/或核的总数量，它不一定等于*当前用户*所能调配的 CPU 数量。例如，在多用户系统中用户能使用的 CPU 数量可能会受到系统的限制。同样地，对于当前的任务，程序检测出的 CPU 数目也不一定就是合理的：用户可能正在同时运行多个 R 进程，而且这些进程本身可能在运行多线程的 BLAS 库、OpenMP 代码或其他底层的并行程序。我们甚至曾经见过 **multicore** 中递归调用的 `mclapply` 函数[4]，这将在 $n = 16$ 核的机器上产生 $2n + n^2$ 个进程。

但到目前为止，`detectCores()` 函数仍是一个有用的指导，它试图检测当前 R 所在机器的 CPU 核数量。在当前所有已知的 R 平台下，它都有相应的方法来完成这一任务。该函数是依赖于操作系统的：针对不同的平台，我们试图报告可用的物理核的数量。

在 Windows 下，该函数默认的选项是报告逻辑 CPU 的数量。对于现代的硬件设备（例如 Intel *Core i7*），这应当是一个合理的参照，因为超线程技术确实能带来显著的性能提升。`detectCores(logical = FALSE)` 所返回的数量是与操作系统的版本相关的：在最近版本的 Windows 系统中，它报告的是物理核的数量，而在较老的版本中，它可能会返回物理 CPU 封装包的数量。

3 类似 apply 的函数

目前为止对 **multicore** 和 **snow** 最常见的应用是将 `lapply`，`sapply`，`apply` 和相关的函数替换为并行的版本。

对于 `lapply`，类似的函数有

```
parLapply(cl, x, FUN, ...)
mclapply(X, FUN, ..., mc.cores)
```

其中 `mclapply` 在 Windows 下不可用[5]，其他的参数在帮助页面中有所讨论。这两个函数在理念上有略微的不同：`mclapply` 会创建 `mc.cores` 个工人，但它们仅仅用于这条语句的计算；`parLapply` 中的工人存在时间更长，通常由 `makeCluster` 创建（同时还指定了集群的大小），并保存在对象 `cl` 中。因此，这种方式下典型的工作流程是

```
cl <- makeCluster(<size of pool>)
# one or more parLapply calls
stopCluster(cl)
```

对于矩阵，软件包中提供了 `parApply` 和 `parCapply` 函数，以及更为常见的 `parRapply`，它是并行化的对矩阵的行进行操作的 `apply` 函数。

4 SNOW 集群

本软件包的大部分代码来自于 **snow** 包并进行了少量的修改，其中的函数同样可以用于 **snow** 包所创建的集群（前提是 **snow** 包位于搜索路径中）。

parallel 软件包中有两个函数可以用来创建 SNOW 集群：`makePSOCKcluster`（一个改进的 `snow::makeSOCKcluster`）和 `makeForkCluster`（Windows 下除外）。它们的区别在于生成工人进程的方式不同：`makePSOCKcluster` 利用 `Rscript` 来启动 R 进程的副本（在相同的主机上或其他地方），而 `makeForkCluster` 则在本机上分流出工人进程（因此它们继承了当前 R 会话的工作环境）。

[4] `parallel::mclapply` 将会检测这一现象，然后以串行的方式运行嵌套的调用。

[5] 一个例外是当 `mc.cores = 1` 时会直接调用 `lapply`。

通常，这两个函数可以利用 makeCluster 函数进行调用。

对于每一个工人而言，stdout() 和 stderr() 都将被重定向，默认的选择是丢弃所有的输出，但它们可以通过 outfile 选项写入日志。注意到上一句话中的 stdout() 和 stderr() 指的是 R 中连接的名称，而不是 C 语言层级的文件句柄。这意味着，正常情况下 R 包中 Rprintf 的输出将被重定向，而 C 语言层级的输出则不会。

用户可以调用 setDefaultCluster() 来设置集群的默认值：当用户调用了诸如 parApply 之类的高层级函数但没有显式指定集群参数时，默认的集群将会被使用。当你准备反复使用某个集群时，需要注意的是这些工人的工作空间会不断积累上次使用后留下的 R 对象，并且某些软件包可能已经加入到了搜索路径中。

如果集群不是创建在当前机器上（“localhost”），那么 makeCluster 可能需要提供更多的参数：

- 如果工人机器的配置与主机器不同（例如，它们可能具有不同的 CPU 架构），那么需要设置 homogeneous = FALSE，并且你可能还需要将 rscript 参数设置为工人机器中 Rscript 所在的完整路径。
- 工人机器需要知道如何与主机器进行通信：一般而言这可以通过主机名来实现（通过 Sys.info() 函数获取），但在私有网络中，情况可能会有所不同，你也许需要为 master 参数设定一个名称或一个 IP 地址，例如 master = "192.168.1.1"。
- 默认情况下 ssh 将被用来在工人进程上启动 R。如果该程序有其他的名称，例如在 Windows 下使用 PUTTY，那么参数应设置为 rshcmd = "plink.exe"。此外，SSH 应被设置为静默验证，如果它需要输入密码，那么它可能无法正常工作。
- Socket 通信使用的端口将从 11000:11999 中随机选择：如果需要使用其他端口，则需要设定 port 参数或环境变量 R_PARALLEL_PORT。

5 分流

除了 Windows 系统之外，**parallel** 软件包还复制了 **multicore** 软件包中带有 mc 前缀的函数，例如 mccollect 和 mcparallel。（**multicore** 软件包中包含了带前缀和不带前缀的两个版本，但不带前缀的函数名很容易被其他软件包屏蔽，例如 **lattice** 软件包中也有一个名为 parallel 的函数。）

parallel 包同样提供了 **multicore** 中的底层函数，但它们没有从命名空间中导出。

此外还有两个高层级的函数 mclapply 和 pvec：与 **multicore** 版本中的不同，本软件包中这些函数会默认使用两个处理器核，用户可以通过设定 options("mc.cores") 来进行调节。当软件包被加载时，这个选项的默认值将从环境变量 MC_CORES 进行读取。（将该选项设置为 1 会禁止并行操作：Windows 下这些函数只是形式上地存在，它们会强制设置 mc.cores = 1。）

从 R 2.15.0 开始，mcmapply 和 mcMap 提供了类似于 mapply 和 Map 的功能。

请留意本文档之前提及的在 GUI 环境中使用分流的注意事项。

在同一次会话中，R 的主进程和分流出的进程享有共同的临时目录 tempdir()，这可能会造成一些问题，因为许多程序假定这个目录对于 R 进程是私有的。此外在 R 2.14.1 之前，两个进程可能会通过 tempfile 在临时目录中选择到相同的临时文件并同时对其操作，而它们并不知道这个文件是否正在被其他进程所使用。

分流出的工人进程与主进程享有共同的文件句柄，这意味着所有工人进程的输出都将定向到主进程中“stdout”和“stderr”所指向的位置。（这并不是在所有操作系统上都成立；当主进程正在从“stdin”进行批量读取时，分流也可能会造成一些问题。）设置参数 mc.silent = TRUE 会关闭子进程的“stdout”输出，而“stderr”不受影响。

共享文件句柄的机制同样会对图形设备产生影响，因为分流出的工人会继承所有已打开的主进程的图形设备。子进程不应修改这些内容。

6 随机数生成

当并行计算中需要利用（伪）随机数时，有一些注意事项应格外小心：运行计算任务的各进程/线程需要运行独立的（最好是可重复的）随机数序列。只要条件允许，一个能避免诸多麻烦的办法是将所有生成随机数的过程都在主进程中完成。在 **boot** 软件包（`1.3-1` 及以后的版本）中，只要有可能就会采用这种办法来生成随机数序列。

当一个 R 进程启动时，它会从已保存的工作空间中读取对象 `.Random.seed` 来设置随机数种子，或者如果之前没有保存工作空间，则会在随机数生成器第一次使用时根据系统时钟和进程 ID 来设置种子（参见 `RNG` 的帮助）。因此，在两种情况下工人进程会具有相同的随机数种子——一是包含了 `.Random.seed` 的工作空间被还原，二是主进程在分流之前已经使用了随机数生成器。如果以上二者皆非，那么工人进程将得到不可重复的随机数种子（但以很大的概率使得所有工人的种子互不相同）。

另一种可选的办法是在主进程中生成一系列的随机数，然后以此作为工人进程的随机数种子，从而使得结果可重复。一般而言这样的做法已经很安全了，但也有人担心这样会使得不同工人上的随机数序列有接近的趋势。对此，一种解决办法是在选取种子时，尽量隔开一定的距离进行选取；然而，在一个随机数序列中，离得远的随机数不一定就比离得近时更加独立。另一种想法（如 **JAGS** 包所使用的）是让每个独立的进程使用不同的随机数发生器。

parallel 包所使用的方法是 L'Ecuyer *et al.* (2002) 的一个实现：这种方法使用单一的随机数生成器，在这个生成器产生的随机数序列中（周期大约为 2^{191}），每隔 2^{127} 步选取一个数作为随机数种子，并生成随机数序列。生成器采用的是 L'Ecuyer (1999) 中的生成器，选择它的原因[6]是它具有相对长的周期和相对小的种子（6 个整数），同时，不像 R 中默认的 `"Mersenne-Twister"` 生成器，这种方法能方便地计算固定步数之后的下一个种子值。生成器的算法如下：

$$
\begin{aligned}
x_n &= 1403580 \times x_{n-2} - 810728 \times x_{n-3} \mod (2^{32} - 209) \\
y_n &= 527612 \times y_{n-1} - 1370589 \times y_{n-3} \mod (2^{32} - 22853) \\
z_n &= (x_n - y_n) \mod 4294967087 \\
u_n &= z_n / 4294967088 \text{ unless } z_n = 0
\end{aligned}
$$

得到的种子即为 $(x_n, x_{n-1}, x_{n-2}, y_n, y_{n-1}, y_{n-2})$，而且 k 步迭代之后 x_n 和 y_n 的系数可以预先进行计算。对于 $k = 2^{127}$，可以通过 R 函数

```
.Random.seed <- nextRNGStream(.Random.seed)
```

来计算 k 步之后的种子值。

从 2.14.0 开始，R 开始支持 L'Ecuyer (1999) 随机数发生器，对应的函数为 `RNGkind("L'Ecuyer-CMRG")`。因此，要实现 L'Ecuyer *et al.* (2002) 中的想法，我们只需运行

```
RNGkind("L'Ecuyer-CMRG")
set.seed(<something>)
## start M workers
s <- .Random.seed
for (i in 1:M) {
```

[6]除了因为这是作者们的共识！

```
    s <- nextRNGStream(s)
    # send s to worker i as .Random.seed
}
```

对于 SNOW 集群，上述代码已经在 `clusterSetRNGStream` 函数中得以实现，并成为 `mcparallel` 和 `mclapply`（默认情形下）函数的一部分。

除了序列（2^{127} 步）之外，还有子序列的概念，它是每隔 2^{76} 步选取一个种子。函数 `nextRNGSubStream` 可以让当前的状态移至下一个子序列。

CRAN 软件包 **rlecuyer** (Sevcikova and Rossini, 2004–present) 提供了一个直接的针对原始（更重量级）C 实现的 R 接口。这个程序针对命名序列进行操作，每个序列都包含了 3 个种子（每个种子由 6 个元素构成）。在 R 中，可以通过适时地保存 `.Random.seed` 来模拟这一特性。**rstream** 包 (Leydold, 2005–present) 提供了另一个利用 S4 类实现的接口。

7 负载均衡

在第一节的简介中我们提及了一种动态分配任务给工人的策略：通常这被称为“负载均衡”，它可以通过 `mclapply(mc.preschedule = FALSE)`，`clusterApplyLB` 和封装函数 `parLapplyLB`、`clusterMap(.scheduling = "dynamic")` 来实现。

当不同任务的计算时间有很大的不同，或不同节点的计算能力相异时，负载均衡就能突显其优势所在，但有一些注意事项需要留心：

1. 随机数序列会分配给节点，所以当节点的任务涉及随机数生成时，它们很有可能是不可重复的（因为任务的分配依赖于节点的工作量）。当然，我们只需一点额外的工作，就可以针对每个任务预先分配好随机数序列。

2. 更加需要注意的是任务的分配。假设有 1000 个任务要分配给 10 个节点，标准的途径是将每 100 个任务的区块分配给一个节点，而负载均衡的方法则是每次将一个任务分给一个节点，这样一来，通信的开销将会很大。所以，合理的选择是让任务数多于节点数，但不超过 100 倍（甚至不应超过 10 倍）。

8 可移植性考虑

对于那些想把并行计算纳入代码中的开发者，他们需要决定在多大程度上程序是可移植并且高效率的：没有一种办法能在所有的平台上都保持最优。

使用 `mclapply` 通常是最简单的办法，但在 Windows 下，这个函数是以串行方式运行的。如果并行计算只是在一台多核的类 Unix 单机服务器上运行，那么这样的设置就已经足够了——这是因为 `mclapply` 只能运行在共享内存的单机系统中。此外，这种方法能向后兼容，如果需要串行，只需要设置 `mc.cores = 1`。

`parLapply` 可以在任何支持 socket 通信的系统中运行，而且可以利用一个实验室内所有空闲的 CPU 核。但即使是在一台机器上，socket 通信也可能被系统拦截，而对于一个实验室内的计算机网络，这种现象将极有可能发生。目前这种方法还无法向串行进行兼容，在未来似乎也不太可能实现（这是因为工人开启了一个与主进程不同的 R 环境）。

`1.3-3` 之后版本的 **boot** 软件包提供了使用以上两种方法的代码实例，它同时还提供了串行版本的代码。

9 扩展案例

在统计学中，粗粒度并行最常见的应用或许就是进行多重的随机模拟，例如大量的 bootstrap 抽样，或多次的 MCMC 模拟。我们为这两类问题分别举一个例子。

注意到有些例子在 Windows 下只能以串行的方式进行，有些则是具有非常大的计算强度。

9.1 Bootstrapping

boot 软件包 (Canty and Ripley, 1999–present) 是 Davison and Hinkley (1997) 所写专著的支持软件。Bootstrap 是简单并行的一个常见的例子，一些计算置信区间的方法都需要上千次的 bootstrap 抽样。**boot** 软件包从 `1.3-1` 版本开始已经在其主函数中内置了并行的支持，但在这里我们将演示如何将原始的（串行）函数进行并行化。

我们考虑两个利用 `cd4` 数据集的例子。`cd4` 来自于 **boot** 包，我们关心的是 CD4 细胞数量前后两次测量之间的相关性。第一个例子是直接的模拟，通常称为*参数 bootstrap*。非并行的版本如下所示：

```
library(boot)
cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
cd4.boot <-
boot(cd4, corr, R = 999, sim = "parametric", ran.gen = cd4.rg, mle = cd4.mle)
boot.ci(cd4.boot, type = c("norm", "basic", "perc"), conf = 0.9, h = atanh,
    hinv = tanh)
```

要利用 `mclapply` 实现并行，我们需要将完整的模拟分成若干次完成，在这里我们将同时运行两轮模拟，每轮进行 500 次重抽样：

```
cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
run1 <-
function(...) boot(cd4, corr, R = 500, sim = "parametric", ran.gen = cd4.rg,
    mle = cd4.mle)
mc <- 2  # set as appropriate for your hardware
## To make this reproducible:
set.seed(123, "L'Ecuyer")
cd4.boot <- do.call(c, mclapply(seq_len(mc), run1))
boot.ci(cd4.boot, type = c("norm", "basic", "perc"), conf = 0.9, h = atanh,
    hinv = tanh)
```

在很多情况下我们都会进行类似的编程：将运算封装到一个函数之中，从而使程序变得简洁。上述代码的倒数第二条语句，正是并行版本的

```
do.call(c, lapply(seq_len(mc), run1))
```

如果希望利用 `parLapply` 来实现并行，我们可以进行类似的操作：

```
run1 <- function(...) {
    library(boot)
    cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
    cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
   boot(cd4, corr, R = 500, sim = "parametric", ran.gen = cd4.rg, mle = cd4.mle)
}
cl <- makeCluster(mc)
## make this reproducible
clusterSetRNGStream(cl, 123)
library(boot)  # needed for c() method on master
cd4.boot <- do.call(c, parLapply(cl, seq_len(mc), run1))
boot.ci(cd4.boot, type = c("norm", "basic", "perc"), conf = 0.9, h = atanh,
    hinv = tanh)
stopCluster(cl)
```

注意到在 `mclapply` 中，所有我们用到的软件包和对象都会自动继承给工人进程，而 `parLapply` 则一般没有这种特性[7]。此外，将计算任务分配给哪个进程也是一个需要考虑的问题：例如，我们可以将 `cd4.mle` 的计算交给工人（如上例），或将其交给主进程然后把返回值传递给工人。对于后者，我们将其代码展示如下：

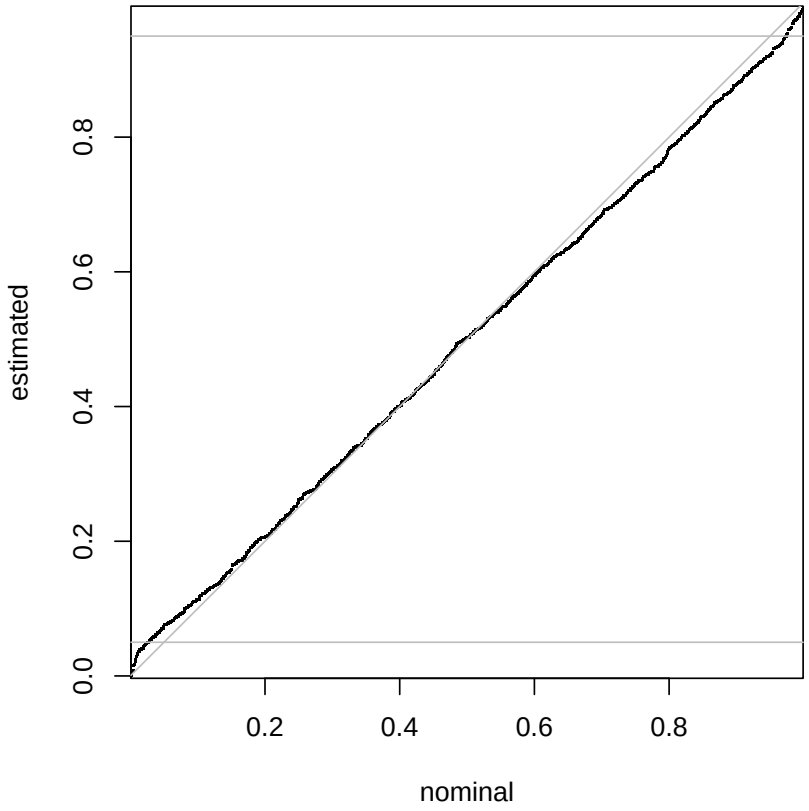
```
cl <- makeCluster(mc)
cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
clusterExport(cl, c("cd4.rg", "cd4.mle"))
junk <- clusterEvalQ(cl, library(boot))  # discard result
clusterSetRNGStream(cl, 123)
res <-
clusterEvalQ(cl, boot(cd4, corr, R = 500, sim = "parametric", ran.gen = cd4.rg,
    mle = cd4.mle))
library(boot)  # needed for c() method on master
cd4.boot <- do.call(c, res)
boot.ci(cd4.boot, type = c("norm", "basic", "perc"), conf = 0.9, h = atanh,
    hinv = tanh)
stopCluster(cl)
```

对于相同的问题，如果采用二重 bootstrap，则其计算量要远大于之前的版本，标准的程序代码是

```
library(boot)
R <- 999
M <- 999  ## we would like at least 999 each
cd4.nest <- boot(cd4, nested.corr, R = R, stype = "w", t0 = corr(cd4), M = M)
## nested.corr is a function in package boot
op <- par(pty = "s", xaxs = "i", yaxs = "i")
qqplot((1:R)/(R + 1), cd4.nest$t[, 2], pch = ".", asp = 1, xlab = "nominal",
```

[7] 一个例外是使用 `makeForkCluster` 创建集群。

```
    ylab = "estimated")
abline(a = 0, b = 1, col = "grey")
abline(h = 0.05, col = "grey")
abline(h = 0.95, col = "grey")
```




```
par(op)

nominal <- (1:R)/(R + 1)
actual <- cd4.nest$t[, 2]
100 * nominal[c(sum(actual <= 0.05), sum(actual < 0.95))]

## [1]  2.7 97.2
```

在一台 8 核的 Linux 服务器上，如果只使用一个核，程序将运行 55 秒。
如果使用 `mclapply`，则代码为

```
mc <- 9
R <- 999
M <- 999
RR <- floor(R/mc)
```

```
run2 <- function(...) cd4.nest <- boot(cd4, nested.corr, R = RR, stype = "w",
    t0 = corr(cd4), M = M)
cd4.nest <- do.call(c, mclapply(seq_len(mc), run2, mc.cores = mc))
nominal <- (1:R)/(R + 1)
actual <- cd4.nest$t[, 2]
100 * nominal[c(sum(actual <= 0.05), sum(actual < 0.95))]
```

该程序使用了服务器上所有的核，共运行了 11 秒（流逝时间）。

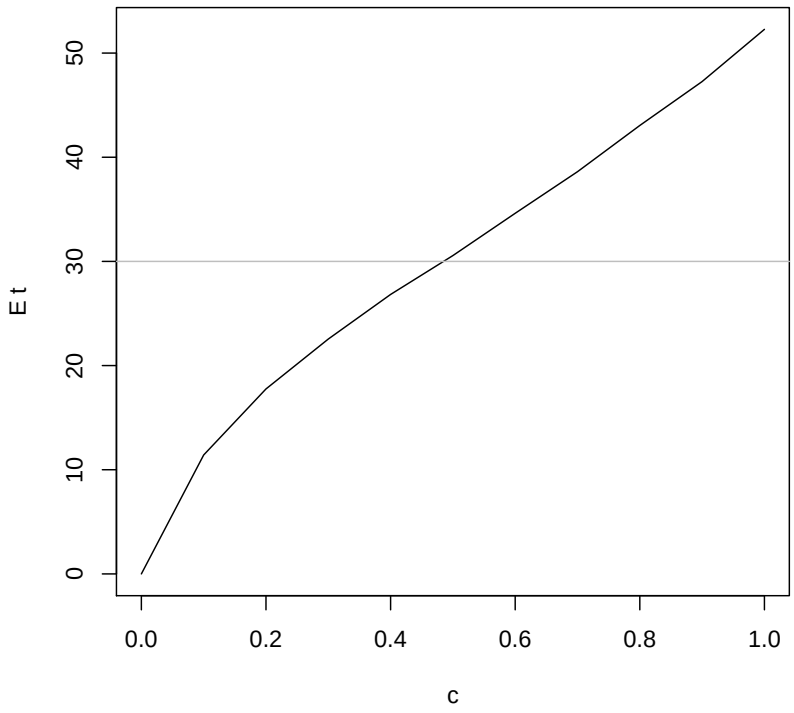
9.2 MCMC 模拟

Ripley (1988) 讨论了 Strauss 过程的极大似然估计，它是矩方程

$$E_c T = t$$

的解，其中 T 是 R-close pairs 的数量，t 是其观测值，在下例中即为 30。一个串行版本的尝试如下所示：

```
library(spatial)
towns <- ppinit("towns.dat")
tget <- function(x, r = 3.5) sum(dist(cbind(x$x, x$y)) < r)
t0 <- tget(towns)
R <- 1000
c <- seq(0, 1, 0.1)
## res[1] = 0
res <- c(0, sapply(c[-1],
    function(c) mean(replicate(R, tget(Strauss(69, c=c, r=3.5))))
        )
plot(c, res, type = "l", ylab = "E t")
abline(h = t0, col = "grey")
```



如今这段代码大约需要运行 20 秒，但当它在 1985 年首次完成时，需要运行许多个小时。一个并行的版本大致为

```
run3 <- function(c) {
    library(spatial)
    towns <- ppinit("towns.dat")  # has side effects
    mean(replicate(R, tget(Strauss(69, c = c, r = 3.5))))
}
cl <- makeCluster(10, methods = FALSE)
clusterExport(cl, c("R", "towns", "tget"))
res <- c(0, parSapply(cl, c[-1], run3))  # 10 tasks
stopCluster(cl)
```

这段程序只运行了 4.5 秒，再加上 2 秒钟的时间用以建立集群。如果使用分流集群（无法在 Windows 下实现），那么程序的启动会更加快速，同时配置也更加简单：

```
cl <- makeForkCluster(10)  # fork after the variables have been set up
run4 <- function(c) mean(replicate(R, tget(Strauss(69, c = c, r = 3.5))))
res <- c(0, parSapply(cl, c[-1], run4))
stopCluster(cl)
```

如你所预料的那样，`mclapply` 的版本会更加简洁：

```
run4 <- function(c) mean(replicate(R, tget(Strauss(69, c = c, r = 3.5))))
res <- c(0, unlist(mclapply(c[-1], run4, mc.cores = 10)))
```

如果你的机器不具有 10 个左右的核，那么你可能会考虑对任务进行负载均衡的处理，因为每段模拟所消耗的时间会随着 `c` 的变化而变化。这可以通过 `mclapply(mc.preschedule = FALSE)` 或 `parSapplyLB` 来实现。而这样的劣势在于结果是不可重复的（在此处无关紧要）。

9.3 软件包安装

当前已经有超过 4000 个可用的 R 包，如果你想进行一次完整的安装，那么将这个过程并行化将是非常有帮助的。在 `install.packages` 中，我们通过并行的 `make` 实现了这一功能，但这种方法可能并不合适。[8]运行的过程中，某一些任务的运行时间可能比其他的长得多，但我们无法事先预知哪些任务是这种情况。

我们在此展示一个用 **parallel** 安装软件包的例子，它目前也用于 CRAN 对软件包进行核查的过程。假设有一个函数 `do_one(pkg)`，它用来安装一个单个的软件包然后返回。接下来的任务就是将 `do_one` 运行在尽可能多的 `M` 个工人上，同时保证在安装 `pkg` 之前，它所有（直接的和间接的）依赖的软件包都已被安装。由于安装单个软件包时可能会屏蔽其他的包，因此我们应允许同时运行的任务数有所变化。下面的代码实现了这一效果，但需要用到一些底层的函数。

```
pkgs <- "<names of packages to be installed>"
M <- 20 # number of parallel installs
M <- min(M, length(pkgs))
library(parallel)
unlink("install_log")
cl <- makeCluster(M, outfile = "install_log")
clusterExport(cl, c("tars", "fakes", "gcc")) # variables needed by do_one

## set up available via a call to available.packages() for
## repositories containing all the packages involved and all their
## dependencies.
DL <- utils:::.make_dependency_list(pkgs, available, recursive = TRUE)
DL <- lapply(DL, function(x) x[x %in% pkgs])
lens <- sapply(DL, length)
ready <- names(DL[lens == 0L])
done <- character() # packages already installed
n <- length(ready)
submit <- function(node, pkg)
    parallel:::sendCall(cl[[node]], do_one, list(pkg), tag = pkg)
for (i in 1:min(n, M)) submit(i, ready[i])
DL <- DL[!names(DL) %in% ready[1:min(n, M)]]
av <- if(n < M) (n+1L):M else integer() # available workers
while(length(done) < length(pkgs)) {
    d <- parallel:::recvOneResult(cl)
    av <- c(av, d$node)
```

[8]首先，系统可能不支持并行的 `make`，此外我们发现有一些软件包无法通过 `make` 正确地安装。

```
    done <- c(done, d$tag)
    OK <- unlist(lapply(DL, function(x) all(x %in% done) ))
    if (!any(OK)) next
    p <- names(DL)[OK]
    m <- min(length(p), length(av)) # >= 1
    for (i in 1:m) submit(av[i], p[i])
    av <- av[-(1:m)]
    DL <- DL[!names(DL) %in% p[1:m]]
}
```

9.4 传递“...”

“...”的语法在并行计算中并不能很好地工作，这是因为在任务发送给工人之前，惰性求值（lazy evaluation）可能会有所延迟。而在分流方法中则不存在这个问题，因为惰性求值所需要的信息都存在于分流出的工人中。

对于类似于 **snow** 的集群，解决的办法是让 `...` 中的 promise 对象[9]强制执行，同时保持所有需要的信息都已被提供。下面是 **boot** 包中 `boot()` 函数的一个演示：

```
fn <- function(r) statistic(data, i[r, ], ...)
RR <- sum(R)
res <- if (ncpus > 1L && (have_mc || have_snow)) {
    if (have_mc) {
        parallel::mclapply(seq_len(RR), fn, mc.cores = ncpus)
    } else if (have_snow) {
        list(...)  # evaluate any promises
        if (is.null(cl)) {
            cl <- parallel::makePSOCKcluster(rep("localhost", ncpus))
            if (RNGkind()[1L] == "L'Ecuyer-CMRG")
                parallel::clusterSetRNGStream(cl)
            res <- parallel::parLapply(cl, seq_len(RR), fn)
            parallel::stopCluster(cl)
            res
        } else parallel::parLapply(cl, seq_len(RR), fn)
    }
} else lapply(seq_len(RR), fn)
```

注意到`...`是 `boot` 的一个参数，所以在运行

```
list(...)  # evaluate any promises
```

之后，它将展开为 `boot` 所在环境中的对象，进而这些对象也将存在于 `fn` 所在的环境中，最后该环境将连同 `fn` 一起发送给工人。

[9]参见 R 帮助文档 *R Language Definition* 的 2.1.8 节——译者注。

10 与之前版本的差别

parallel 软件包对并行随机数发生器的支持与 **snow** 不同，而 **multicore** 不支持并行随机数发生器。

10.1 与 multicore 的区别

multicore 精心编写了一系列代码来使得 `R.app` 的 Aqua 事件循环和 `quartz` 图形设备的事件循环能继承给子进程。在 **parallel** 中，取而代之的是在 R 的执行程序中为子进程做一个标记。

`detectCores` 在计算物理 CPU 数量时会有偏差，这是由 Sparc Solaris 下的严重问题引起的，在该系统中 **multicore** 会计算得出荒谬的进程数。

函数 `fork` 和 `kill` 加上了 `mc` 前缀，并且没有被导出。这避免了和其他软件包的冲突（例如 **fork** 软件包），同时注意到 `mckill` 并不如 `tools::pskill` 通用。

parallel 包不再提供别名函数 `collect` 和 `parallel`。

10.2 与 snow 的区别

snow 设置的超时超过了 POSIX 所要求的最大值，并且没有提供为工人设置超时的办法。这会导致 Solaris 下的进程互锁。

`makeCluster` 通过调用 **snow** 来生成 MPI 或 NWS 集群。

`makePSOCKcluster` 被加以改进，因为 **parallel** 包在所有系统中的位置都是已知的，而且目前 `Rscript` 程序在所有系统下都可用。对工人的日志记录被设置为追加到文件，所以多重进程都可以被记录到日志中。

`parSapply` 的用法已经与 `sapply` 保持一致。

`clusterMap()` 加入了 `SIMPLIFY` 和 `USE.NAMES` 参数，这使得它成为了并行版本的 `mapply` 和 `Map`。

计时接口没有复制到 **parallel** 中。

参考文献

Canty A, Ripley BD (1999–present). "`boot`: Bootstrap Functions." URL http://cran.r-project.org/package=boot.

Davison AC, Hinkley DV (1997). *Bootstrap Methods and Their Application*. Cambridge University Press, Cambridge.

L'Ecuyer P (1999). "Good parameters and implementations for combined multiple recursive random number generators." *Operations Research*, **47**, 195–164. URL http://www.iro.umontreal.ca/~lecuyer/myftp/papers/combmrg2.ps.

L'Ecuyer P, Simard R, Chen EJ, Kelton WD (2002). "An object-oriented random-number package with many long streams and substreams." *Operations Research*, **50**, 1073–5. URL http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf.

Leydold J (2005–present). "`rstream`: Streams of random numbers." URL http://cran.r-project.org/package=rstream.

Ripley BD (1988). *Statistical Inference for Spatial Processes*. Cambridge University Press, Cambridge.

Sevcikova H, Rossini T (2004–present). "`rlecuyer`: R interface to RNG with multiple streams." URL http://cran.r-project.org/package=rlecuyer.

Tierney L, Rossini AJ, Li N, Sevcikova H (2003–present). "Simple Network of WorkStations for R." URL http://www.stat.uiowa.edu/~luke/R/cluster/cluster.html.

Urbanek S (2009–present). "`multicore`: Parallel processing of R code on machines with multiple cores or CPUs." URL http://cran.r-project.org/package=multicore.