An introduction to Data Science and Big Data

Fabrice Rossi

December 18, 2017

Contents

1 General Introduction

1.1 Big Data?

1.1.1 What is Big Data for you?

Introductory interactive discussion about Big Data.

1.1.2 Objectives of the course

- intellectual self-defense
- clearing the confusion around big data
- when do you **not** need big data solution
- a tour of the most common solutions

1.2 The Big Data confusion

The terms "Big Data" should be used only to refer to very large data sets and related applications. It should not be used to describe predictive modeling *in general*. This confusion is damaging because solutions designed for super large data sets are generally not adapted to normal size data sets.

1.2.1 Doug Laney's definition

Analyst at META group (now Gartner). Proposed in 2001 the 3 V's:

1. Volume: data size
2. Velocity: streaming context
3. Variety: text, image, video, etc.

Sometimes complemented by Veracity (data quality, confidence in the results). The really important points are Volume and Velocity. Variety has been around for ages and Veracity is addressed in other contexts. Keeping the 3/4 V's is a manifestation (among others) of the confusion.

1.2.2 Consequences of the confusion

Very high volume data sets cannot be stored, queried and processed by standard solutions (e.g. on a single yet powerful computer). Thus specialized solutions have been developed. Main example: the Map Reduce paradigm by Google, Hadoop as open source solution.

However, they have a quite high entry cost, both in human terms and in processing power terms. In other words, those tools are *not* adapted to small to medium scale data sets.

Standard confusion story:

1. we want to do predictive modelling but we confuse that with big data
2. we read tutorials and books about big data finding discussions about predictive modelling, reinforcing our confusion
3. we use the dominant open source solution, hadoop
4. performances are very bad so we add more computers, reinforcing our belief that we do big data magic
5. wash, rinse, repeat

1.2.3 What has changed?

Are there really a before and an after? an emergence of Big Data?

This is indeed the case with the massive use of computers to handle every day life. Computer use produces *traces* (a.k.a. activity log). The most basic example is maybe the logs of web servers which contain:

- the ip of the client (your computer) -> this gives *many* facts about you such as your location and your internet provider;
- the request date and time;
- the requested page;
- a user agent (a description of your web browser);
- the referrer (if available, this is where you come from).

Numerous other examples can be given. They all revolve around the following facts:

1. data collection is easy via internet services (web sites and specialized protocols and apps)
2. there is an incentive to provide accurate personal information (e.g. on facebook when discussing with your family and friends, on whatsapp/telegram where you give your phone number, on amazon where your provide your coordinates, etc.)

3. most of the providers are US based and do not have to satisfy very stringent data privacy rules (contrarily to e.g. in Germany or in France)

4. computer resources are *super cheap*, for instance storage:

   - in 1956 the first HDD by IBM costed 50 k $ (equivalent to roughly 430 $ K of 2013) and stored 5 MB
   - in 1995 a typical 1 GB HDD costed 850 $ (equivalent to 1300 $ in 2013)
   - in 2005 a typical 200 GB HDD costed 200 $ (equivalent to 120 $ in 2013)
   - in 2010 a typical 1 TB HDD costed 80 $ (equivalent to 85 $ in 2013)
   - in 2013 a typical 2 TB HDD costed 110 $
   - the cost has dropped from 10000 $ per MB to 6.33 cents per GB!

5. computer resources are *available* especially via cloud computing systems (and their ancestors like grid computing systems)

Notice that while internet is major provider of personal data, big data appear in other contexts, such as:

- pre-internet services (e.g. credit/debit card, air company frequent flyer programs, etc.)
- physics: the Large Hadron Collider (LHC) and its ancestors
- biology:
  - in 2001, the estimated cost of sequencing the full human genome was estimated to have been around 2 billions of $ for 15 years of work
  - in 2009, it costed 100 k $ for 1 year of work
  - in 2014, it cost 1000 $ for 24 hours of work
- smart grids: ERDF linky
- the open data movement

1.3 Big?

A.k.a. Volume.

1.3.1 Some orders of magnitude for storage

See this wikipedia entry.

1. one bit: 0 or 1
2. one byte: 8 bits, 0 to 255 (for instance)
3. four bytes (32 bits): integers, old memory address space, IPv4 address, single precision floating point number
4. eight bytes (64 bits): long integers, current memory address space, double precision floating point number
5. sixteen bytes (128 bits): IPv6 address, minimum length of private keys in state-of-the-art symmetric encryption algorithms
6. 640 kB ($640 \times 2^{10} \times 8$ bits): typical size of a book written in English (500 pages, 2000 characters per page, compressed representation), maximum memory size in the original IBM PC standard
7. 4 MB ($4 \times 2^{20} \times 8$ bits): typical size of a mp3 encoded song
8. 10 MB: typical size of a jpeg encoded high quality image
9. 40 MB: typical size of a raw image from a high end DSLR
10. 650 MB: capacity of a compact disk
11. 2 GB ($2 \times 2^{30} \times 8$ bits): Iphone 6s main memory size
12. 3 GB: Galaxy S6 main memory size
13. 4 GB: maximum addressable memory on a standard 32 bits architecture
14. 4,7 GB: capacity of DVD (single layer, single side)
15. 8 GB: typical main memory size of a high end laptop / medium range desktop
16. 16 GB: typical main memory size of a workstation
17. 25 GB: capacity of a Blu-ray disk
18. 128 GB: typical main memory size of a 16 core server
19. 200 GB: maximal capacity (in 2015) of a micro SD card
20. 512 GB: maximal capacity (in 2015) of a SD card, typical capacity of a SSD
21. 1 TB ($2^{40} \times 8$ bits): typical size of a HDD

22. 10 TB: maximal capacity (in 2015) of a HDD

23. 16 TB: maximal capacity of a SSD (just announced)

24. 300 PB ($300 \times 2^{50} \times 8$ bits): capacity of Facebook data warehouse in April 2014, corresponds to 300 000 1 TB HDD or to 1.2 millions of BR disks

25. 15 EB ($15 \times 2^{60} \times 8$ bits): estimated capacity of Google data warehouse in 2013 (unconfirmed but realistic), corresponds to 15 millions of 1 TB HDD or 600 millions of BR disks

1.3.2 Computational capabilities

Expressed in flops: floating point operations per second. There are essentially two ways of doing floating point calculations: with a CPU (general purpose microprocessor) and with a GPU (graphical processing unit, specialized originally for graphics).

Current Intel architecture (as of 2015):

- Haswell and Broadwell microarchitecture
- commercial names: core i3, i5, i7 and M; pentium and celeron; xeon
- from 2 to 18 cores (4 is the standard)
- clock frequency between 2 GHz and 3.5 GHz
- roughly two operations per tick per core
- around 100 Giga flops per CPU for general calculation for the best ones (around 7000 $)
- specific faster operations for linear algebra
- can address up to 1.5 TB of memory

Graphical processing units

- harder to program
- less memory, up to 16 GB per GPU
- enormous flops: around 1.5 Tera flops per GPU for the best ones (e.g. Nvidia Tesla K40, around 3000 $)

1.3.3 An example of theoretical cost analysis

Let us consider the case of some regression analysis on a data set with N observations in dimension P (P variables). The calculation is dominated by a matrix multiplication ($X^T X$) and the inversion of the resulting matrix. The computational cost is in $O(NP^2 + P^3)$. Memory occupation is in $O(NP + P^2)$. In general $N > P$ and thus the mixed terms dominate the costs.

P	P^3 time in seconds with CPU	P^3 time in seconds with GPU
10	1 (-8)	6.6666667 (-10)
100	1 (-5)	6.6666667 (-7)
1000	0.01	6.6666667 (-4)
10000	10.	0.66666667
100000	10000.	666.66667

P	P^3 time in hours with CPU	P^3 time in hours with GPU
100000	2.7777778	0.18518519
1 (6)	2777.7778	185.18519

P	P^3 time in days with CPU	P^3 time in days with GPU
1 (6)	115.74074	7.7160494
1 (7)	115740.74	7716.0494

Of course, 1 million of features is quite large, so a more reasonnable maximum value for P might be 10000. With such a high value of P, one needs many examples to build reliable predictive models and thus N should be at least of the order of 100 000. Then we have the following timings.

N	P	$NP^2 + P^3$ s (CPU)	$NP^2 + P^3$ s (GPU)
1000	10	1.01 (-6)	6.7333333 (-8)
5000	100	5.1 (-4)	3.4 (-5)
10000	1000	0.11	7.3333333 (-3)
50000	1000	0.51	0.034
100000	1000	1.01	0.067333333
1 (6)	1000	10.01	0.66733333
1 (8)	1000	1000.01	66.667333

This analysis shows that the number of features is really the limiting factor (at least in this example). A natural question is whether the number of features can reach very high values.

1.3.4 Number of features

A very large number of features is natural in many contexts:

- images: high resolutions DSLR have several millions of pixels
- text mining: a text can be seen as a count vector in a very high dimensional space (one dimension per word)
- genome mining: roughly 3 billions of bases, with 0.1% of variability between humans (so 3 millions of differences...)
- log data: key typed, web page browsed, electricity consumption, position, etc.

1.3.5 A warning about parallelism and distribution

It should be noted that the computational capabilities given above are

1. difficult to achieve without very complex programs which make use of specialized instructions (from MMX in 1996 to recent AVX/FMA) and respect the cache hierarchy;
2. based on intrinsically parallel programs.

I use the terms parallel program to refer to programs designed for multiple cores or at least a unique computer with possibly several CPUs. I use the terms distributed program for programs that are designed to run on multiple loosely connected machines. The main point of distinction is whether memory (RAM) is shared in a efficient way or not.

In order to reach good performances on a high end machine, one needs to develop parallel programs. To achieve good performances on set of computers, one needs to develop distributed programs.

1.4 Take home message

Big Data should be used to refer to very large data sets and applications dealing with them. Technological and sociological changes have allowed the collection of enormous data sets. While modern CPU/GPU have enormous processing capabilities, some standard methods do not scale to a large number of features. Very high feature counts can be found in numerous applications. Parallel processing is mandatory to reach high performances.

2 Big Data Needs and Applications

I'm separating the data storage part from the data management/querying part, while I'm grouping data generation and data acquisition. This is somewhat different from some surveys, e.g. cite:HuWenEtAl2014TowardScalable.

2.1 Data Generation and Acquisition

The Big Data "revolution" is based on massive data collection capabilities which are needed because of the massive data generation/production this is done daily. This is enabled by several important elements:

1. data are produced essentially in digital format, nothing is analog anymore (music, image, video, text)
2. data collection is either automated or crowdsourced
3. mobile connected devices enable generation and collection from everywhere
4. part of the mobile evolution simplifies even further point 1, e.g. voice recognition, music indexing, etc.

The ongoing evolution will lead to even more data via the "internet of things", i.e. connected devices ranging from activity trackers to "smart" thermostats.

Data acquisition is generally part of the generation process, for instance a connection to a web site generates a log entry in the web server (ditto for e.g. e-commerce). For some applications, data acquisition can prove complex because of the needed transmission rates from distributed sensors (internet of things) and servers.

Both generation and acquisition are simplified by local processing. For instance, activity trackers detect automatically the type of activity (sitting, standing, etc.) and can therefore transmit only time intervals. Music recognition (a.k.a. Shazam) is based on the local calculation of a signature.

Problems in a nutshell

1. scalable data collection (enough servers)
2. scalable data transmission (enough pipes to the servers)

Those can be solved via e.g. cloud offers.

Big Data Design Patterns

1. "crowdsourcing": make sure data are produced as part of the service, preferably by the users
2. distributed computation: summarize the local data using local resources (signature, etc.) before sending them

2.2 Data Storage

Collecting data is nice, but they must be stored somewhere. It should be noted that while the *streaming* model has been studied a lot, in practice data get stored. They might not be usable readily, especially for short latency applications, but they are still stored somewhere.

As pointed out above, storage is now very cheap. There are 3 levels of storage, in increasing order of latency, capacity and cheapness:

1. RAM
2. SSD
3. HDD

In order to give the illusion of a very large storage capacity with low latency at a decent price, a hierarchical scheme is used with caching. Perhaps more importantly, storage is abstracted from the hardware via high level **file systems** and/or via high level systems (NAS) or lower level ones (SAN). The key idea is to separate computational resources from storage resources and to connect them via a (local) network. This allows, among other things, to add more and more storage capacity (this is part of **horizontal scaling**).

Price examples in 2015

- super high end SAN, Dell PowerVault MD3820f:
  - with 24 TB of hyper fast HDD: 20 000 €
  - with 40 TB of super fast HDD: 32 000 €
- high end SAN, Dell PowerVault MD3800i:
  - with 48 TB of hyper fast HDD: 20 000 €
- high end NAS, Synology RS18016xs+:
  - with 48 TB of high end HDD: 8 700 €

NAS and SAN are **integrated** solutions which can be replaced by basic off the shelf PC with local storage and a **distributed file system**. We will discuss this aspect further when we will talk about the hadoop ecosystem.

Problems in a nutshell It is generally considered that this aspect of big data is mature enough to provide off the shelf solutions adapted to most of the real word situations (as in the case of collection and acquisition, via e.g. cloud computing offers).

2.3 Data Management and Querying

Stored data are useless if they cannot be manipulated. The standard way of doing that is to use a relational database management system (RDBMS). The so-called NoSQL movement provides alternative solutions.

On the one hand, RDBMS seem to be the best solution, notably via ACID transactions: they should provide the properties needed in order to ensure that parallel data production, access, modification, etc. do not break the whole system. On the other hand, the CAP "theorem" says that they can't provide what's needed, thus alternative should be explored.

2.3.1 Relational database management system

They have three major elements:

1. they are based on the relational model which represents interlinked data in a way that both avoids redundancy in the data storage and embeds logical constraints on the data
2. they provide ACID transactions:

   Atomicity all or nothing

   Consistency valid states only

   Isolation serial

   Durability completed transactions are done, like forever
3. almost all RDBMS implement a variant of SQL

2.3.2 The CAP "theorem"

The CAP "theorem" is a conjecture (or a general observation/principle) made by Eric Brewer in 1998-2000 about distributed systems (see cite:Brewer2012CAP12YearsLater for a recent point of view of Brewer himself on the principle). The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

consistency (C) equivalent to having a single up-to-date copy of the data (*this is not the C of ACID*);

high availability (A) we can access data in a reasonable time;

tolerance to network partitions (P) if we break up the system into pieces that can't communicate, it continues to work "correctly".

Unfortunately, this cannot be a theorem because of the vagueness of the properties. Some theorems have been proved by interpreting the properties in specific theoretical contexts, but the overal principle remains a form of principle.

That said the principle is very simple, especially viewed on the partition point of view (as in cite:Brewer2012CAP12YearsLater). Indeed if a distributed system is partitioned, it contains at least 2 parts that are not connected at a certain point. Then if a query arrives to one part, this part cannot decide if is has the latest available version of the data. Thus, it has either to refuse the query (breaking A) or to risk inconsistency (breaking C).

Because traditional RDBMS operate under the ACID principle, they cannot be partition tolerant. One of the basis of the NoSQL movement is then to move away from the RDBMS model in a tentative to be more tolerant to the partition issue. As pointed out in cite:Brewer2012CAP12YearsLater, the actual issue is whether *during a partition event* one should favor consistency or availability, and then how to recover a fully consistent system when the partition disappears.

2.3.3 NoSQL

Another way to approach the CAP "theorem" consequences is to replace ACID by other properties. The most popular ones are the BASE ones:

Basically Available availability is the main goal;

Soft state no formal definition (again!). It could mean that the state can change without user intervention (to reach consistency);

Eventually consistent if updates stop, the system will eventually become consistent.

BASE has been the driving design philosophy of NoSQL DBMS. However, it should be noted that the NoSQL term is well chosen but poorly understood. In fact, the common point of most of the NoSQL DBMS is that they do not use the relational model but other forms of organizations (which are not adapted to the SQL language, hence the NoSQL acronym).

This does not mean that all NoSQL DBMS are distributed. In particular, the key-value database model, which is a typical non relational database structure, can be implemented as a local system or in a distributed fashion.

Nevertheless, there are numerous large scale distributed data store that can be used in big data contexts, both to store the data and to query them, albeit in a limited way compared to SQL possibilities. There are also data store dedicated to some types of data, for instance for documents (XML, JSON, etc.) and for graphs.

2.3.4 Problems in a nutshell

The database aspect of big data is a very active research area. The movement has been originally to move away from traditional relational DBMS to the NoSQL distributed DBMS. However, many of them offer little or no consistency guarantee, leading to e.g. data loss. Thus, many real world solutions are based on hybrid systems that combine relational DBMS for crucial data with NoSQL distributed DBMS for other data. The current trend in research seems to be the NewSQL class, exemplified by Google's Spanner. It's a move

back to almost relational systems with SQL interface and strong consistency guarantees. At a smaller scale, distributed versions of standard RDBMS seem to work quite well (e.g. MySQL Cluster).

2.4 Data Mining

Obviously, one of the main goal of the Big Data approach is to extract information from the pile of data collected. The application range is enormous and contains, among many others:

- system monitoring (dashboard): this is very important for large web sites, for instance, and more generally in order to understand how the users are actually using the service (see the retweet interface added by Tweeter, for instance)
- user modeling for:
  - recommendations: amazon, netflix, etc.
  - targeted advertisement: google, facebook, etc.

In addition to those obvious applications, it should be noted that the data are also used to build models that are in turn used to offer services to users. For instance (recommendation is in between):

- content searching (recommendation outside of the commercial context)
- machine translation
- face detection/recognition (in picture)
- automatic summarizing
- etc.

The main difficulty of data mining in the context of big data is the need for distributed programming. Real word data mining is mostly tailor made and cannot generally be done via graphical user interfaces (GUI). Some form of programming is generally needed. Thus one of the main research work that has been done in the context of big data mining was to provide simple yet powerful data mining framework that hide the underlying complexity of distributed systems. The most famous one is the Hadoop ecosystem.

2.5 Take home message

Handling Big Data is complex. The processing/value chain of Big Data starts with production/acquisition, followed by storage (low level aspects), then management and querying and finally mining. The first two steps are now handled via rather mature solutions. The database step (the third one) is still under heavy research with no obvious

solution, even if the NewSQL paradigm seems to provide the best of the RDBMS world with the scalability of the NoSQL world. The data mining step is under heavy research and has to face the tension between allowing easy development of specialized data mining tasks and the utter difficulty of programming efficient and sound distributed algorithms.

3 Shared memory parallel programming

3.1 Introduction

The **only way** to deal with large to big data is to use some form of parallel processing. There is simply no computational unit that is able to deliver the number of flops that is needed for modern data handling. Unfortunately, parallel processing is difficult. I discuss first the easiest situation, the case of shared memory parallel programming (see cite:HerlihyShavit2008ArtOfMultiprocessorProgramming for an excellent textbook on efficient concurrent data structures).

3.2 Shared memory hardware structure

A computer is made of processing units (PU) and of memory (among other things). Current computers (even phones) have multiple PU (the cores) that all use the same memory. This is the standard shared memory structure. Sharing is done at the **hardware** level which guarantees good performances, at least in theory. Software level shared memory is also available, but it comes with a higher programming cost and with lower performances when there is no direct hardware support.

3.2.1 Threads, processes and memory

The simplest way to use shared memory is via the thread model. The general unit for a program is a *process* which regroups several execution contexts (the *threads*) and a unique memory context. Each process runs under its unique virtual memory space, which is mapped by the Memory Management Unit (MMU) to the physical memory. This is completely transparent for the programmer. Threads have access to the full virtual memory space, leading to a fully shared memory structure with hardware support.

3.2.2 MMU and caches

The MMU is responsible off the virtual to real mapping. It also handles its caches (the Translation lookaside buffer). This cache and the CPU caches are crucial to reach high performances because the cheap big memory has enormous latency (somewhat mitigated by a very high bandwidth). I cannot cover the details in this course (see cite:Drepper2007Memory and my course in French), but even getting single threaded high performance code is difficult because of the caches. Some numbers, the typical latency of a memory request at each level of the cache (intel):

- register (core): 1 cycle
- L1 (core): 3 cycles
- L2 (core): 12 cycles
- L3 (CPU): 38 cycles
- local RAM (CPU): 65 ns (130 cycles at least)
- non local RAM (NUMA, see below): 105 ns (40ns from QPI)

3.2.3 Non-uniform memory access

There is a limit to the number of cores that can be integrated on a single CPU (18 for the current Intel architecture). When a server needs more PU, it has to integrate several CPU. In order to maintain the shared memory principle in this case, two solutions exist.

The simplest one consists in having a unique shared memory for all the processors and interconnection bus to which the processors send requests for memory access. This does not scale well to a large number of CPU as the bus becomes a contention point.

A more advanced solution the Non-uniform memory access (NUMA) principle. The main idea is that each CPU has its own memory. When a process uses more memory than the one attached directly to its preferential CPU, it will use memory from the other CPUs transparently (via the MMU). Of course, access to the distant memory is slightly less efficient than access to the local memory. This explains the idea of a preferred CPU.

3.3 Programming difficulties

There are essentially two issues with parallel programming on shared memory architectures:

1. the correction issue: is a parallel program really producing the same results as its sequential version?
2. the efficiency issue: does the parallel program make a good use of the hardware by reducing the user time needed to perform a task?

The first issue is very complex and cannot be solved without hardware guarantees, such as locks and related tools (in particular compare and set primitives). The key point is to enable an a priori unbounded number of threads to agree on a common value.

The second issue is somehow even more annoying in the sense that solving the first issue with obvious solutions (such as locks) leads frequently to inefficient solutions: they do work but the user time (the wall clock time) is not significantly reduced compared to a sequential program, while in theory one could expect the wall clock time to be divided by the number of threads (as long as each thread maps to a core).

While those two problems are extremely important, they should be overemphasized in the context of machine learning algorithms, for instance. Many machine learning

algorithms have "embarrassingly parallel" structures in the sense that the result is obtained by combining a series of almost independent calculations (this is the case of resampling methods, for instance). In fact, the difficulties outlined above are a consequence of *coupling* between the calculations, either via some shared data structures (a database) or because of important communication needs.

Let consider for instance the base problem of computing a distance matrix between N points in $\mathbb{R}^P$. The square distance between two points is calculated as the sum of squared differences and involves $3P$ operations. This has to be done for $M = \frac{N(N-1)}{2}$ pairs, so the total sequential cost is $\frac{3PN(N-1)}{2}$. Obviously, distances can be computed independently. If we number the distances from 1 to $M = \frac{N(N-1)}{2}$, we can then split the calculation on T threads which will be responsible of compute M/T distances. No communication will be needed between the threads and we can expect the wall clock time to be divided by the number of threads.

3.4 Standard API

In order to program for shared memory systems, one needs some Application Programming Interface (API) that allows to either to manipulate threads and locks (low level API) or to express that some parts of the program can be executed concurrently (high level API). High level languages such as Java and C# provide both levels, including very high level tools that mask to some extent the difficulties outlined above.

In particular the very popular **Fork-Join** paradigm is readily available. In this paradigm a master thread forks into several tasks (which are mapped transparently to threads), wait for them to terminate, and collects the results in a sequential joint phase. For instance in the distance calculation above, the master thread would load the data set and forks on the pairwise distance calculations themselves.

For C/C++, the standard solution is **OpenMP**. This is an API integrated into compilers and their runtime libraries. It is based on pragmas of the form `#pragma` (this is a standard way to introduce compiler level extensions in C). Examples of pragmas include:

`#pragma omp parallel` this creates additional threads to execute the current construct

`#pragma omp parallel for` this will execute the subsequent for loop over multiple threads

`#pragma omp barrier` this corresponds to the join part of the fork join paradigm, all threads will wait for each other before passing the pragma

OpenMP is very declarative in the sense that explicit fine management of the underlying threads is not really needed (and is not possible). An example from Tim Mattson follows: the program computes π by integration (based on $\arctan 1 = \frac{\pi}{4}$ and the property $\arctan(z) = \int_0^z \frac{1}{x^2+1} dx$).

3.4.1 Sequential program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i = 0; i < num_steps; i++){
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    printf("%g\n",pi);
}
```

3.4.2 A parallel version in OpenMP

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
#define NUM_THREADS 4
int main ()
{
    double pi;
    int nthreads;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int i, id,nthrds; double x, sum;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i = id, sum = 0.0; i < num_steps; i = i + nthreads){
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
        sum = sum*step;
#pragma atomic
        pi += sum;
    }
```

```
    printf("%f\n",pi);
    return 0;
}
```

3.5 Take home message

Even in the case of shared memory, parallel programming is very difficult: getting quickly correct results is very challenging. The problem revolves around communications between parts of the program: it's easy to lock everything (to get correct results) but then efficiency suffers a lot. Fortunately, many problems can be handled with some simplified approaches: the fork-join principle that underlines OpenMP and the data parallelism that appears in `foreach`. The latter principle is particularly adapted to many learning tasks, for example.

4 Distributed systems

4.1 Introduction

When the data cannot be handled efficiently on a single computer, one has to rely on a *distributed system* which consists in several computers interconnected by a network. The main difference between a distributed system and a multi-core/CPU system is that in the former, the memory is not shared directly between threads in a single process. On the contrary, several processes are running (on different computers) and communicate with messages (even if this communication can be implicit when using high level API). Notice that distributed computing is sometimes reserved for tightly coupled machines (with very fast network for instance) and opposed to *cluster computing* where the machines can be loosely coupled. I tend to avoid this distinction.

The main advantage of distributed systems over shared memory systems is the reduced risk of inconsistent behavior induced by the explicit aspect of sharing (which is done via message passing). However, this is probably the only advantage as distributed systems are very difficult to master and have significant overhead compared to SMP systems. As such, numerous API have been proposed in order to ease programming (to some extent).

At the low level, programming for a distributed system means having some way of knowing the structure of the system (e.g., how many computing nodes are available) and some communication means. At a higher level, one can hope to describe a complex task as a set of simple tasks with dependencies and then let a framework turn this description into a fast distributed program (this could apply to SMP).

4.2 Some API and programming models

- API:

  Parallel Virtual Machine (PVM) PVM is the ancestor of parallel programming API designed for distributed systems. It was created in the late 80's and is based

on message passing. It is still used but as been somehow superseded by MPI.

Message Passing Interface (MPI) MPI is standard created in the 90's. It focuses on message passing and can be seen as a communication protocol.

- programming paradigm:

  Map Reduce Map Reduce is a quite simple paradigm which was popularized by Google in 2004 (cite:DeanGhemawat2004MapReduce). It was initially the main computational model for the Hadoop framework. It suffers from major performance problems in many situations.

  Spark Spark is one of the recent paradigms that address Map Reduce limitations.

4.3 Message Passing Interface

MPI (Message Passing Interface) is the current standard for high performance computing. It has commercial and open source implementations that are generally optimized for some hardware. Interestingly, MPI implementations are sufficiently optimized to be used also for shared memory computers. In fact the first version of MPI was limited to message passing, but starting from the second version (in the late 90's), distributed memory operations are available. They can easily be implemented as shared memory operations.

The API of MPI is very rich and of quite low level. It can be seen as the distributed equivalent of low level thread API for SMP systems. A MPI program consists in several processes that are mapped to possibly different computers. The API allows processes to communicate in various ways:

- point to point direct communication
- broadcasting (one to all)
- reducing (all to one)
- etc.

Communication is used to synchronize processes via rendez-vous (as part of point to point communication) and barrier (global rendez-vous). Communications can be organized in a quite convenient way with some form of virtual topology.

The API corresponds to a library to which some runtime tools are associated. As in the case of threads based programs (e.g. openMP), one starts a unique program that is replicated on the different machines that form the cluster. The runtime configuration can be quite complex as the user must have access to all the machines used in the distributed system. Some form of distributed identification has therefore to be in place. In addition, having a shared file system is recommended. Finally, firewall can introduce a lot of problems and it is recommended to avoid using filtering inside the cluster.

4.4 Map reduce

Map reduce is a simple parallel processing paradigm proposed by Google in 2004 (cite:DeanGhemawat2004
The novelty does not lay in the idea of separating computation between map operations and reduce operations (this is fairly standard and was available in 1995 in MPI) but rather in the underlying implementation.

4.4.1 High level principles

The base idea of Map Reduce is to decompose a computational task into map operations and reduce operations:

map the map operation is applied to some local data by each computational node

reduce the reduce operation is applied to groups of data produced by the map operations

More specifically, map reduce operates on indexed data. Each data point consists in a pair (key, value). The map operation takes one pair and outputs a list of pairs. The reduce operation takes one key and a set of values associated to that key, resulting from map operations. It outputs a list of values (also with keys).

The classical example is the word counting one:

- the map operation takes a document as input (with the key of the document, for instance its title) and outputs a list of pairs (word, 1);
- the reduce operation takes as input a word (keys from the map step) and **all** associated values and outputs the sum of the obtained values.

This can be obviously improved by having the map operation to count words. This can be somewhat automated by a Combiner functionality in Google's map reduce.

4.4.2 Some examples

Co-buying assume we have shopping carts (orders) given as sets of items. The goal is to count the pairs of items (for correlation analysis). Two solutions

1. simple pairwise:
   - map each set to a series of 1 keyed by item pairs
   - reduce on pairs
2. hashmap based:
   - map each set to a series of hashmaps. Each hashmap maps items to 1 and is keyed by another item
   - reduce emits pairs

Conditional statistics assume we have records with two variables: one integer (or nominal) variable X and one numerical variable Y. The goal is to compute some statistics on Y conditionally on X. It could be for instance the average revenue based on age. A possible solution:

- map each record to the value of Y keyed by X
- reduce on X and compute the statistics

Naive Bayes classifier assume we have records with one discrete target variable Y and p discrete explanatory variables X_1 to X_p. To compute the NBC probabilities we need the number of occurrences of values of Y as well as the number of join occurrences of values of Y and of X_p. A possible solution:

- map each record to:
  - $((y, *, *), 1)$: to count the number of occurrences of $Y = y$
  - $((*, *, *), 1)$: to count the number of records
  - $((y, p, x_p), 1)$: to count the number of cooccurrences of $Y = y$ and $X_p = x_p$
- reduce on the keys by summing all the values of a given key and emitting the sum, e.g., $((y, *, *), k)$ for the number of occurrences of $Y = y$.

4.4.3 Design principle

While Map Reduce is frequently presented as a general purpose paradigm for distributed system (something that it is not), its main interest resides in the features of its implementations that are quite different from what is expected for a typical MPI cluster, for instance. In addition, its design is quite specific and in a way is taking a very different path from the one taken by MPI. One could say that MPI targets computationally intensive tasks while map reduce is more adapted to data intensive tasks. In particular, map reduce is adapted to situations where each datum is processed once while MPI is well suited for iterative algorithms, for instance. Map reduce is also designed to take advantage of local storage while MPI is generally built on the idea of a distributed storage.

In practice Map reduce is designed to be run on a very large number of machines that provide a distributed storage. Each machine should be aware of what it stores locally. When a map reduce job is run on the cluster, maps are performed locally: each machine is given a copy of the map task and will run it on the data is stores. Then a *shuffle* operation allows partial results to be shared and transferred from machine to machine in an optimized way. Reduce operations are handled by idle machines using the shuffling facility.

It should be noted that in theory, map reduce could be run on some computers that do not implemented a distributed storage. However, the main advantage of map reduce is the automated leveraging of locality which cannot be done if the computational backend is not aware of the fine details of the storage backend.

As work distribution and communications are handled by the map reduce framework, a typical implementation can be fault tolerant. The original Google use case targeted a

loosely integrated cluster of standard computers with a standard network. In such a cluster, machine failure is common, a fact that motivated this fault tolerance feature.

4.4.4 Hadoop MapReduce

Hadoop is an open source framework for distributed storage and distributed processing. It provides in particular:

Hadoop Distributed File System (HDFS) distributed storage component

Hadoop MapReduce distributed processing

As explained above, Map reduce is only efficient if it has some knowledge of data location. This is why the execution engine of Hadoop is tightly coupled with its storage engine.

Hadoop is implemented in Java and provides therefore a native Java API for MapReduce. This is basically done by implementing a Mapper interface and a Reducer interface. The main way to use other languages instead of Java is via the Hadoop Streaming approach which provides a basic text based interface.

4.5 Spark

Spark is an advanced processing paradigm that overcomes many of the limitations of map reduce. One of the main idea of Spark is the introduction of the notion of a *working set* of data on which operations are conducted. This eases the implementation of both iterative methods and interactive analyses, while improving a lot their performances over a map reduce implementation.

Spark is implemented in Scala, a very high level functional and object oriented programming language. Spark has also a Java and a Python interface that are quite close to the Scala one. It offers also an official R interface.

4.5.1 Resilient Distributed Datasets (RDD)

The concept of working set is implemented by the Resilient Distributed Dataset (RDD). A RDD is a read only collection of objects that are distributed on the underlying cluster. In general, a Spark program start by getting a RDD from a distributed storage and then produces a series of RDD via transformation operators. A very interesting aspects of RDD is that they can be kept in the main memory of the computers of the cluster (this is called *caching*) which generally improves (a lot) the performances.

The term *resilient* refers to the fact RDD can be recomputed from their starting point in case of a machine failure.

4.5.2 Operations on RDD

A spark program consists in performing operations on RDD. Typical operations include:

mapping apply a transformation to each object of the RDD to built another object (in another RDD)

flatMap similar to map in map reduce (man one object to possibly many objects)

join the join operation from data bases

None of those operations is actually performed on the fly during the program execution. On the contrary, they are only recorded. The actual computation takes place when an *action* is performed on a RDD. This enables optimizing the series of operations. Actions include:

count the size of a RDD

collect allows to iterate on a RDD

reduce accumulation function

A word count example:

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

5 Big Data and R: Data Management and Querying

This section is dedicated to data manipulation in R at the RDBMS level. I review different levels of manipulation from importing to actual in memory manipulation. A somewhat dated source on this topic is provided by an official R manual. A better source is the Efficient R Programming book (chapter 5 about input/output and chapter 6 about data carpentry).

5.1 Elementary level (importation)

The basic tool for data "manipulation" in R is the `data.frame` format coupled with the `read.table` function. The former is the standard way of representing data tables (a.k.a. rectangular grids) in R while the later enables users to load spreadsheet like data from text files.

While this pair is convenient and easy to use, it has major problems, especially when dealing with large scale data (even medium scale data, in fact). In particular, users tend to confuse the need for *importing* data into R with the need for *storing* data in a R workflow. While importing data, reliance on text files is sometimes mandatory. However, once the data has been imported in R, there is no reason to rely on text files, apart for exporting data to e.g. another program.

We focus here in the importation aspect as it is important for interaction with non R world, but keep in mind that using the RDS native format is the best solution for R only processing.

5.1.1 Limitations of `read.table`

The `read.table` function and its friends, `read.csv`, `read.delim`, etc. are inherently limited by the text format on which they rely. Limitations include:

- non fixed data format (separator, quotes, numerical encoding, etc.);
- no meta data (what is the type of a data column?);
- encoding issues between platforms.

Notice than variations of the CSV format should be avoided (use the RFC and `write.csv` when exporting). However, in our context, the biggest problem is the inefficiency of the function which can be very slow at loading even medium size data.

For instance, reading a medium size matrix as in the following code takes roughly 110 seconds (on a Xeon CPU E3-1271 v3 @ 3.60GHz (4 cores) with a standard hard drive and 16 GB of main memory).

```
a <- matrix(rnorm(5000*5000), 5000, 5000)
write.csv(a,file="demo-a.csv",row.names=FALSE)
a <- as.matrix(read.csv("demo-a.csv"))
```

Notice in addition that the file itself is large: 450 MB against around 190 MB when in memory. Moreover, the function uses a very amount of memory during its execution. For instance, when only the reading operation is performed above, the peak use of memory is 2.5 GB (notice the occupation is not a consequence of converting to a matrix).

As a comparison reading from the binary representation as in the following code takes only 1 second on the same computer. In addition the file uses 190 MB as expected and the memory usage is only 230 MB, as expected.

```
a <- matrix(rnorm(5000*5000), 5000, 5000)
saveRDS(a,file="demo-a.Rds")
a <- readRDS("demo-a.Rds")
```

There are ways to improve `read.table`. A few tricks can be use at the `read.table` level. For instance one can use the `colClasses` parameter to specify the nature of the columns as a way to reduce memory usage. Overestimating the number of rows also helps (`nrows` parameter). Finally removing the possibility of comments by setting the `comments` parameter to `""` speeds up loading. However, those improvements are somewhat minor. For instance the following code runs in 60 seconds but still uses around 2.5 GB of memory.

```
a <- as.matrix(read.csv("demo-a.csv"),colClasses="numeric",
               nrows=6000,comment.char="")
```

The canonical solution in plain R consists in using the `scan` function. It's a low level function but it's much more adapted to reading large matrices, for instance. In our context, `scan` can be used as follows.

```
a <- matrix(scan("demo-a.csv",skip=1,what=numeric(0),n=5000*5000,
                 sep=","),
            5000,5000,byrow=TRUE)
```

This code runs in 12 seconds and uses 430 MB of memory. While this uses quite a lot of resources compared to the binary format solution, the improvement over the naive use of `read.table` is significative.

5.1.2 Alternatives to `read.table` and `scan`

Fortunately, importing text files into R can be done more efficient functions than `read.table` and `scan`. They come from external packages.

1. Readr The readr package is specifically designed to read CSV files and related format. A simple way to use it is given in the following snippet:

   ```
   a <- read_csv("demo-a.csv")
   ```

   This takes around 24 seconds to read the matrix with a memory usage of 500 MB. As in the case of `read.csv` and friends, the efficiency of the function can be improved by providing hints to it such as the type of the columns and the size of the resulting data frame. For instance the following code runs in only 10 seconds (and uses 450 MB of memory).

   ```
   a <- read_csv("demo-a.csv",
                 col_types=cols(.default=col_double()))
   ```

   In fact, most of the overhead of the base call comes from reading a lot of lines in order to guess the structure of the columns. A very simple way of reducing the overhead is to tweak the `guess_max` parameter to a small number of lines (default is 1000). That might fail on complex data, but on numerical matrices, it works nicely. For instance, the following code runs in 10 seconds.

   ```
   require(readr)
   a <- read_csv("demo-a.csv",guess_max=10)
   ```

   Notice than `read_csv` outputs a R object that is slightly richer than a data frame. For instance a column specification can be extracted from such an object with the `spec` function (that can be useful for reading several files with a similar format).

2. Data.table Another solution is provided by the data.table package. The main goal is this package is not importing data into R, but it offers the `fread` function which load efficiently CSV files. This function is efficient by default. For instance, the following code runs in 5 seconds and uses 240 MB of memory.

   ```
   require(data.table)
   a <- fread("demo-a.csv")
   ```

3. Discussion For normal CSV files, the best solution is `fread` from `data.table`, by far. On large files (several GB), `fread` and `read_csv` (from `readr`) are roughly equivalent, while the latter can be quite slow on very small files (with no consequences). The only limitation of those fast methods is that they might fail to detect or to report inconsistencies in the data format because they base their analysis on the first lines of the file.

5.2 In memory manipulation

Obviously, importing data into R is only the first step of data analysis. In order to perform data mining, one needs to manipulate the data performing was it is sometimes called *data carpentry*. The goal is basically to prepare the data for data mining, by reorganizing them, cleaning them, etc. Many of those tasks can be done via simple R scripts. However, there are much more efficient solutions provided by several packages, especially by dplyr and data.table.

One should be particularly careful about R's tendency to copy data structures, for instance columns in data frames. On small data, that's not an issue, but on large ones, that can become very problematic. As a consequence, an unwritten rule about R is to avoid working with data structures that are larger than roughly 20% to 33% of your available memory.

5.2.1 Running example

We work here with a simple data set from governmental open data, the HM Land Registry Price Paid Data which contains land property transactions in the UK. The 2016 data is of a reasonable size, roughly 1 millions observations, while the full data set starts in 1995 and contains currently almost 23 millions of transactions (with year 2016 included).

We want to perform on those data very standard data carpentry such date formatting and all sorts of basic aggregate calculations. Those calculations are presented in the next section with the basic plain R implementation.

5.2.2 Basic approach

As in the case of data importation, the basic approach relies on data frames and on associated functions. Notice in particular that R includes numerous "data base like" functions, such as `aggregate` which resembles the `GROUP BY` SQL statement and `merge` which is comparable to the `JOIN` clauses.

Surprisingly, with minimal precautions, the 2016 data file can be read rather quickly (a few seconds) in R with the base `read.csv` function as follows:

```
pp2016 <- read.csv("data sets/pp-2016.csv",
                   colClasses=rep("character",16),
                   nrows=1040000, comment.char="",
```

```
                    stringsAsFactors=FALSE,
                    header=FALSE)
```

In this data set, the second column gives the price paid for a property, while the fourth column gives the post code of the property. A basic data manipulation would be to ask for the average price paid per post code. In plain R this can be done via the `aggregate` function.

```
pp2016$V2 <- as.integer(pp2016$V2)
price.per.pc <- aggregate(V2~V4,data=pp2016,mean)
```

This takes roughly 16 seconds. As this is a quite high resolution, one can aggregate the price over other geographical zones. For instance, variable number 14 gives the county which the coarser aggregation level directly available in those data.

```
price.per.county  <- aggregate(V2~V14,data=pp2016,mean)
```

Here the result is obtained almost instantaneously because of the small number of counties.

Another natural question concerns the evolution of the prices through time. The third variable contains the date of the transaction but its loaded in a text format. So we need first to convert the text into R `Date` objects and then extract from those objects different variables useful for aggregating the transactions at a different time resolutions.

```
pp2016$Date <- as.POSIXct(pp2016$V3)
pp2016$Month <- months(pp2016$Date)
pp2016$yday <- as.integer(format(pp2016$Date,"%j"))
price.per.month <- aggregate(V2~Month,data=pp2016,mean)
price.per.yday <- aggregate(V2~yday,data=pp2016,mean)
```

All those operations run quickly and the whole processing time is only of a few seconds.

However, the whole data set consists in more than 20 times more data than the 2016 data and thus the limitations of bare R appear:

- reading the full file with `read.csv` takes 210 seconds and uses 8 GB of memory (this is a 3.7 GB file and the internal representation uses 5 GB);
- aggregating over the post codes takes around 40 seconds;
- aggregating over the counties takes around 12 seconds;
- the date conversion is very slow (120 seconds), increases the memory consumption to 11 GB and finally breaks R during subsequent operations.

This last problem can be circumvented by using the `IDate` facility of `data.table`, as follows (for the 2016 case):

```
require(data.table)
pp2016$Date <- as.IDate(pp2016$V3)
pp2016$Month <- month(pp2016$Date)
pp2016$yday <- yday(pp2016$Date)
```

However, the memory usage remains enormous, with peaks around 12 GB for any "complex" operation carried on the data. When the consumption peaks slightly higher, the R process can be killed on a 16 GB computer. When things remain under control, monthly aggregation takes around 13 seconds while the day of year oriented one takes slightly less time around 12 seconds.

Thus base R is more limited here by memory issues than by pure performance problems.

5.2.3 data.table

The data.table package has been designed to ease *data carpentry* in R by introducing SQL like facilities in the data frame model. In addition, the package is very efficient at handling large data, both during loading (as demonstrated before) and during processing.

Loading the short data set is almost instantaneous (1.5 seconds) and uses only a limited memory. Notice that the loaded object is both a data frame and a data table.

```
require(data.table)
pp2016 <- fread("data sets/pp-2016.csv",header=FALSE)
```

A very efficient aspect of `data.table` is its ability to avoid useless data copies via a reference semantics that enables in place modification of a data table. Here, we use it to translate the values in the second columns to integers.

```
pp2016[,V2:=as.numeric(V2)]
```

The key point is the use of the := operator which tells R to update the column rather than to copy it as would be the case using the standard = operator.

Moreover, `data.table` implements a general indexing/aggregating facility that is summarize by the following construct:

```
DT[i, j, by]
```

in which `DT` stands for a data table. In this construction `i` is a selection operator that filter the data table based on some condition. Then `j` stands for some operation to perform on the selected rows, for instance an aggregation operation. And finally `by` corresponds to a grouping of the results according to some criterion.

This can be used to compute the mean transaction price aggregated over post codes as follows;

```
pp2016[,mean(V2),by=V4]
```

Here, we use no filtering/selection, then we compute averages over variable `V2` while grouping values based on variable `V4`. This implementation is very fast as the result is obtained almost instantaneously (contrast with 40 seconds in the `aggregate` case).

In order to do date based processing, we add columns to the data table using the reference semantics:

```
pp2016[,`:=`(Date = as.IDate(V3))]
pp2016[,`:=`(Month = month(Date), yday = yday(Date))]
```

Then we can use the data table main feature to compute the aggregates

```
price.per.month <- pp2016[,mean(V2),by=Month]
price.per.yday <- pp2016[,mean(V2),by=yday]
```

Results are obtained instantaneously. It's also very easy in this case to filter by whatever interesting criteria (this is also possible in `aggregate` via the subset parameter). For instance, the Greater London county has the highest transaction price and one might be interested by their monthly evolution. This is obtained as follows:

```
pp2016[V14=="GREATER LONDON",mean(V2),by=Month]
aggregate(V2~Month,mean,data=pp2016,subset=pp2016$V14=="GREATER LONDON")
```

Both calculations are quite fast on this data set, but the data table one is (10 times) faster and arguably much more readable.

Moving on to the full data set:

- the full data takes 160 seconds without any tricks (plain call to `fread`, and uses 5.3 GB);
- aggregating over the post codes takes around 3 seconds (compare to 40 seconds with `aggregate`);
- aggregating over counties takes less than 1 second;
- date conversions are reasonably fast (26 seconds) and do not increase the peak memory usage of 8 GB (5.7 GB after garbage collection);
- aggregating over month and day of the year are almost instantaneous and, more importantly, do not break R!
- in addition, more complex aggregations are fast and reliable. For instance aggregating over two variables (month and county) is very fast with data table while it breaks R because of a too large memory consumption with `aggregate`

5.2.4 dplyr

As data.table, dplyr has been designed to ease data manipulation in R but also to speed them up. Dplyr follows data table in providing filtering, aggregation and other high level operations inspired by the SQL model. It operates on extended data frames, the tibbles. The companion package `readr` can be used to read directly tabular data into the tibble format. For our running example, this gives

```
require(readr)
pp2016 <- read_csv("data sets/pp-2016.csv",col_names=FALSE)
```

Reading is quite fast for the small data set (8 seconds) and do not waste memory (461 MB). In addition, `read__csv` guesses correctly the type of the columns, especially dates, something that is not done by `fread`.

In order to aggregate with a `GROUP BY` like approach, one used `group_by` in `dplyr`. In our running example, this gives

```
require(dplyr)
pp2016_by_pc <- group_by(pp2016,X4)
```

Once this operation has been done (this take around 7 seconds on the small data set), one can apply other `dplyr` operations: they will respect the grouping structure. For instance the average value per postal code is obtained as follows

```
price.per.pc <- summarise(pp2016_by_pc,price=mean(X2))
```

This operation is instantaneous on the small data set. The county level summary can be obtained very quickly by combining directly the two operations as follows.

```
price.per.county  <- summarise(group_by(pp2016,X14),price=mean(X2))
```

As data table, `dplyr` can easily add columns to an existing data frame. We use it here to add the temporal columns.

```
pp2016 <- mutate(pp2016,Month=months(X3),
                 yday=as.integer(format(X3,"%j")))
```

Then we run the standard aggregation calculations.

```
price.per.month <- summarise(group_by(pp2016,Month),price=mean(X2))
price.per.yday <- summarise(group_by(pp2016,yday),price=mean(X2))
```

Moving on to the full data set:

- the full data takes 170 seconds uses 5.3 GB (with a peak to 11 GB!);
- aggregating over the post codes takes around 19 seconds;
- aggregating over counties roughly 1 second;

- date conversions are fast (12 seconds) and increase the peak memory usage of 8 GB (6.2 GB after garbage collection);
- aggregating over months takes 5 seconds and over days of the year 1 second;
- in addition, more complex aggregations are relatively fast and reliable. For instance aggregating over two variables (month and county) takes 7 seconds.

5.2.5 Summary

We summarize in the following table the run time of different operations on the full data set.

Operation	Plain R	dplyr	data.table
Data loading from CSV	210s	170s	160s
Post code aggregation	40s	19s	3s
County aggregation	12s	1s	<1s
Date conversion	Impossible	12s	26s
Month aggregation	Impossible	5s	<1s
Day aggregation	Impossible	1s	<1s
Month + County	Impossible	7s	<1s

Obviously, data table and dplyr are much better solutions than plain R. data table seem to perform better than dplyr, but the latter as advantages in other contexts. In addition, dplyr can work on top of data tables, which somehow brings together the best of both worlds.

6 Big Data and R: Shared memory parallel programming

This section is dedicated to the use of shared memory systems in R.

6.1 Implicit parallel programming in R

The simplest way to use parallel processing in R is to rely on implicit parallelism, that is to leverage libraries/packages that have been designed to use multiple cores. The main example of this is linear algebra (matrix and vector).

Because of the importance of linear algebra in mathematics, standards have been designed in order to provide efficient API and associated implementations. The standard low level API is BLAS. It provides the elementary operations around matrices and vectors (dot product, matrix vector product, etc.). Numerous efficient implementations of BLAS are available. Most of them are designed for multicore CPU and can reach the theoretical peak performances of those CPU. Two open source very efficient implementations are well known:

1. ATLAS
2. GotoBLAS

R can be configured to use any BLAS implementation instead of its integrated one. This can bring tremendous improvement in computation time for programs that make heavy use of linear algebra. For instance consider the very simple following code:

```
a <- matrix(rnorm(5000*5000), 5000, 5000)
b <- matrix(rnorm(5000*5000), 5000, 5000)
c <- a%*%b
```

It runs in roughly 80 seconds on a Xeon CPU E3-1271 v3 @ 3.60GHz using the single threaded BLAS from R (this is a 4 cores CPU). With Atlas in single threaded mode, the program runs in around 20 seconds. With parallel Atlas the program runs in 6 seconds (with less than 3 seconds in the matrix multiplication). The theoretical peak for general purpose calculation is at 28.8 10^9 flops while the computational load is of 250 10^9 operations, leading to an observed value of almost 84 10^9 flops. Larger matrices lead to even better flops. It turns out that recent CPU include specific instructions that are highly efficient for matrix computation. The CPU used in this test is from the Haswell family which can do up to 16 operations on double precision real number per core and per cycle. Thus the theoretical peak for matrix calculation is around 230 10^9 flops which explains the observed performances. The latest Intel CPU family can perform up to 32 operations per core and per cycle.

6.2 Explicit parallel programming in R

When a program does not rely heavily on linear algebra, implicit parallel programming is not sufficient and one has to rely on explicit parallel programming. The core facility for this is provided by the `parallel` package. However, parallel is rather low level and its simpler to use the `foreach` package that provides a programming interface that is quite different from the fork-join principle. The `foreach` principle belongs to the *data parallelism* model (same code different data) rather than to the *task parallelism* model (different codes).

6.2.1 The π example: base R code

Let's first translate the C code into R:

```
## loop version, very slow
compute.pi.loop <- function(nsteps) {
    sum <- 0
    step <- 1.0/nsteps
    for(i in 1:nsteps) {
        x <- (i-0.5)*step
        sum <- sum + 4/(1+x^2)
```

```
    }
    step * sum
}
```

This is not idiomatic (and in addition quite slow), so we move on to a more idiomatic version.

```
## idiomatic version much faster
compute.pi.idiomatic <- function(nsteps) {
    sum(4/(1+(((1:nsteps)-0.5)/nsteps)^2))/nsteps
}
```

This version is very fast (by R standard) but be warned that it allocates a lot of memory. We obtain the following running time.

`nstep`	10^7	10^8
loop	8s	83s
idiomatic	0.11s	1.2s

6.2.2 Foreach version

The foreach loop construct in R is quite different from the standard for loop in the sense that it does not *iterate* over things but rather apply a block of code to different values. For instance the result of

```
library(foreach)
foreach(i=1:5) %do% i^2
```

is a list containing the following values:

$$1 \quad 4 \quad 9 \quad 16 \quad 25$$

The values of variables in the block are set via the parameters of the `foreach` call while the results are aggregated via a combination function, for instance as in:

```
foreach(i=1:5,.combine=sum) %do% i^2
```

with returns 55. Then the π calculation example can be done with `foreach` as follows:

```
compute.pi.foreach.loop <- function(nsteps) {
    step <- 1.0/nsteps
    sum <- foreach(i=1:nsteps,.combine=sum) %do% {
        4/(1+((i-0.5)*step)^2)
    }
    step * sum
}
```

Unfortunately, this is amazingly slow: 2.4 seconds for `nstep` $= 10^4$. This is because `foreach` is optimized for large tasks while here each task is a very basic calculation. The correct solution consists in wrapping the idiomatic calculation into a `foreach` loop, for instance as follows:

```
compute.pi.foreach.idiomatic <- function(nsteps,ntimes) {
    step <- 1.0/nsteps
    foreach(i=1:ntimes,.combine=sum) %do% {
        sum(4/(1+((seq(i,nsteps,by=ntimes)-0.5)/nsteps)^2))/nsteps
    }
}
```

Then the effects of the `foreach` remain very small as long as `ntimes` is also small, as shown in this timing table.

`nstep`	10^7	10^8
`ntimes` $= 1$	0.37s	3.76s
`ntimes` $= 10$	0.37s	4.2s
`ntimes` $= 100$	0.42s	4.1s
`ntimes` $= 1000$	0.62s	3.8s

6.2.3 Parallel foreach

The main interest of `foreach` is that its calculation can be done in parallel. One has only to replace the `%do%` construction by a `%dopar%` construction and to load a parallel backend. So the modified code is

```
compute.pi.foreach.idiomatic.parallel <- function(nsteps,ntimes) {
    step <- 1.0/nsteps
    foreach(i=1:ntimes,.combine=sum) %dopar% {
        sum(4/(1+((seq(i,nsteps,by=ntimes)-0.5)/nsteps)^2))/nsteps
    }
}
```

The standard parallel backend on linux and MacOS is `doMC` which is registered as follows:

```
# load the library
library(doMC)
# register the backend
registerDoMC()
```

Then, one can test the parallel version. The natural value for `ntimes` is here the number of cores, but it can be interesting to use higher counts. Indeed with `nstep` $= 10^8$ and `times` $= 4$ (on a 4 cores CPU), we get a run time around 1.4s slightly *larger* than the sequential code. But the sequential code cannot run with `nstep` $= 10^{10}$ because it allocates too much

memory (75 GB!) and thus the calculation has to be cut in several parts. This is provided by `foreach` in the idiomatic design which can be run with `nstep` $= 10^8$ and e.g. `times` $=$ 100. This takes approximately 116s (as expected).

6.2.4 Foreach do and don't

The `foreach` package is very useful to bring concurrency to R in a quite simple way. It has additional features that we will study latter in the course. However, it can also be misused as shown above.

- things to avoid:
  1. do not use `foreach` for short tasks
  2. do not nest `foreach` loops without using the specialized feature
- things to do:
  1. optimize the R code before going to foreach
  2. test the `foreach` code with a sequential backend first
  3. test different numbers of cores
  4. leverage the combine facility when possible
  5. to maximize portability use `doParallel` rather than `doMC` but be aware that performances under windows are inferior than under Mac OS and linux.

7 Big Data and R: Distributed Systems

This section is dedicated to the use of distributed systems (no shared memory) in R.

7.1 MPI and R

There are essentially two packages that provide some direct access to MPI under R, `Rmpi` and `pdbMPI`. Both are intended to allow one to program for a MPI cluster with R and thus do not build complex abstraction over it. Programming in R for MPI is arguably simpler than in C because in the latter one has to define e.g. types for messages. In addition part of the communications is automatically handled by the R wrappers.

A very basic example with `Rpmi` is given below:

```
library(Rmpi)
## starts 8 R processes that will execute in parallel
mpi.spawn.Rslaves(nslaves=8)
## ask each slave to execute some code
## mpi.comm.rank() returns the number of the slave
```

```
mpi.remote.exec(mpi.comm.rank()^2)
## quit R after shutting down the slaves
mpi.quit()
```

As seen in this simple example, the `Rmpi` API provides a simple way to execute code on the remote processes and therefore a `foreach` like approach is always possible.

In fact, there is a `doMPI` backend for `foreach` that allows to leverage a MPI cluster directly with `foreach` without modifying the R code. However, because of the operational complexity of MPI, additional code will be needed. For instance, there is not direct `registerDoMPI` function. One needs for instance to perform the following initialization and cleanup code:

```
library(doMPI)
cluster <- startMPIcluster(count=8)
registerDoMPI(cluster)
## the main code goes here
closeCluster(cluster)
mpi.finalize()
```

7.2 Map Reduce and R

There are several ways to use a Hadoop cluster from R:

RHIPE this package belongs to the Tessera framework (which contains 2 other packages, datadr and Trelliscope). The interface of RHIPE is very similar to the original Java interface from Hadoop. It provides utility functions to get inputs from the Hadoop cluster and to send results to it. The central piece of code is provided by a map expression (written in R) and a reduce expression (which can be complemented by some pre and post processing expressions).

RHadoop this is a collection of five R packages which provide an interface to Hadoop. The map reduce package uis `rmr`. The interface provided by RHadoop is arguably of a slightly higher level than the one from RHIPE. In particular, `rmr` uses standard R functions rather than expressions. The expected interface for `map` functions is of the form `function(k,v)` (for key `k` and value `v`), while the one for `reduce` function is of the form `function(k,vv)` again for key `k` but for a list of values `vv`. The function `keyval` is used to output results from the map and reduce functions. A simple example:

```
library(rmr2)
results <- mapreduce(input = "/user/fabrice/census-income.data",
                     input.format =
                         make.input.format(format="csv",
                                           sep=",",
```

```
                                    dec="."),
          map = function(k,v) keyval(v[,1],v[,6]),
          reduce = function(k,vv) keyval(k,mean(vv)))
```

HadoopStreaming this package offers a much more bare bone interface to Hadoop map reduce. It eases the development of R scripts that can then be used in a Hadoop cluster through the streaming facility (the one that allows using something else than Java to write map reduce jobs). No abstraction of map reduce is actually included in this package.

Notice than none of those packages helps administering the Hadoop cluster.

7.3 Spark and R

Spark has an official R interface, provided by the `SparkR` package (and associated tools). SparkR has its own version of the data frame concept, `DataFrame`, which represents in R a RDD. In the SparkR context, R scripts consist mainly in transformation operations on `DataFrame`, using operations that are comparable to the one available in Spark.

```
sc <- sparkR.init()
sqlContext <- sparkRSQL.init(sc)
df <- createDataFrame(sqlContext, mydataframe)
# accessing to a column
select(df, "colonne")
# filtering the data frame
filter(df, df$bidule < 20)
# computing on columns
df$truc <- df$bidule * df$machin
# aggregating
avg(groupBy(df,"departement"))
```

```
sc <- sparkR.init()
textFile <- textFile(sc,"data.txt")
words <- flatMap(textFile,
                 function(line) {
                     strsplit(line, " ")[[1]]
                 })
wordCount <- lapply(words,
                    function(word){
                        list(word, 1L)
                    })
counts <- reduceByKey(wordCount, "+", 2L)
output <- collect(counts)
```

8 References

bibliographystyle:alpha bibliography:~/work/research/bibliography/references.bib