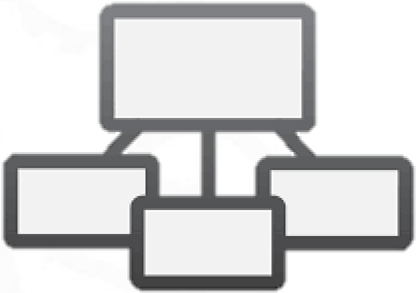
Parallel Computing



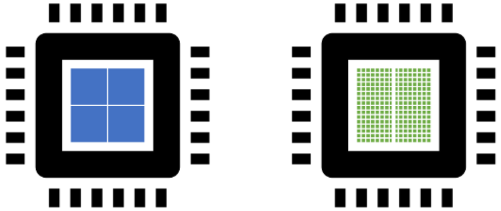
Contents

Parallel Computing

Introduction

Popularization of parallel computing in the 21st century came in response to **processor frequency** scaling hitting the power wall. Scaling the processor frequency is no longer feasible after a certain point due to **power consumption** and **overheating**. Therefore, parallelizing systems and software is the only way to increase processing power.

Importance of parallel computing continues to grow with the increasing usage of multicore processors and **GPUs**. GPUs work together with CPUs to increase the throughput of data and the number of concurrent calculations within an application.

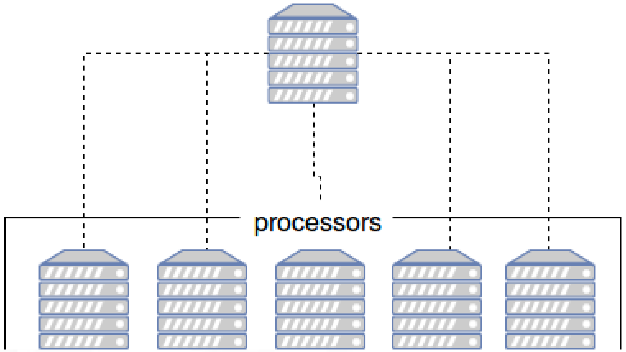


CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

Introduction

Parallel computing refers to the process **of breaking down larger problems** into **smaller, independent**, often similar **parts** that can be executed simultaneously by multiple processors communicating via shared memory. Results are combined upon completion as part of an overall algorithm.

The **main goal** is to increase the computing power available for **faster application processing** and troubleshooting **in exchange for** a **higher computational load** on the system.

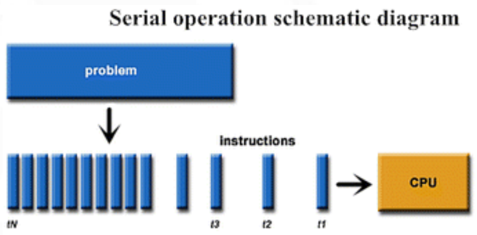


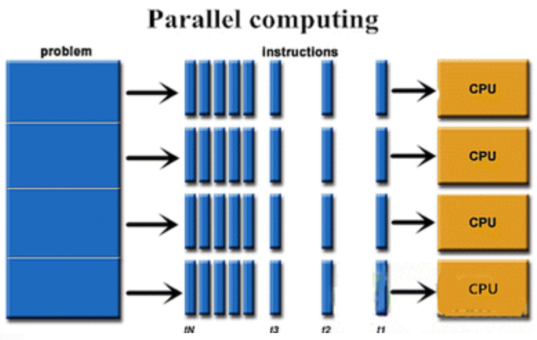
Introduction

- Sequential vs Parallel computing

In **serial computing** a problem is broken down to a series of instructions, which are executed sequentially one after another on a single processor.

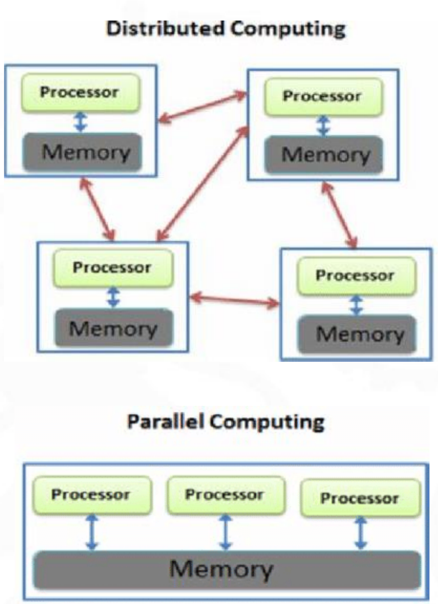
In **parallel computing** a problem is broken into discrete parts that can be solved concurrently, which are broken down to a series of instructions, which are executed sequentially one after another on different processor, all of this controlled by a coordination mechanism.

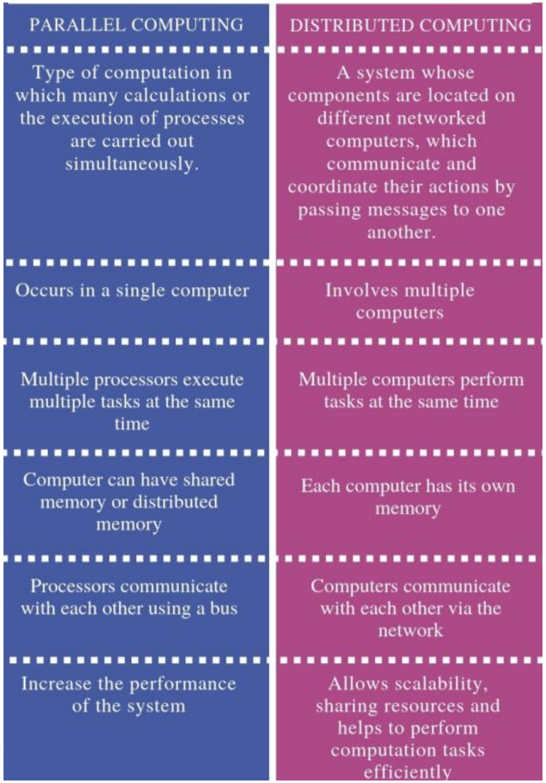




Parallel Computing
Introduction

- Distributed vs Parallel computing





PARALLEL COMPUTING	DISTRIBUTED COMPUTING
Type of computation in which many calculations or the execution of processes are carried out simultaneously.	A system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.
Occurs in a single computer	Involves multiple computers
Multiple processors execute multiple tasks at the same time	Multiple computers perform tasks at the same time
Computer can have shared memory or distributed memory	Each computer has its own memory
Processors communicate with each other using a bus	Computers communicate with each other via the network
Increase the performance of the system	Allows scalability, sharing resources and helps to perform computation tasks efficiently

Parallel Computing
Fundamentals

Parallel processing (in the extreme) means that all the processes **start simultaneously** and run to completion on their own. The potential **speedup** of an algorithm on a parallel computing platform is given by Amdahl's law:

$$S = \frac{1}{1 - p - \frac{p}{s}} = \frac{1}{\frac{p}{N} + X}$$

Where `S` is the potential speedup, `s` is the speedup latency of the execution of the parallelizable part of the task, `p` is the percentage of execution time of the whole task concerning the parallelizable part of the task before parallelization, `N` the number of workers and `X` the percentage serial part of the code. If the non-parallelizable part of the code is 10% of the task, then p=`0.9`.

This law applies to fixed-size problems where the total amount of work to be done in parallel is independent from the number of processors, usually giving **optimistic non-realistic results for large problems**.

Parallel Computing
Fundamentals

In practice, as more computing resources become available the time spent in the **parallelizable part often grows much faster than** the inherently **serial work**. In this case, **Gustafson's law** gives a less optimistic and more realistic assessment of parallel performance:

$$S = 1 - p + sp$$

Gustafson's law assumes that the total amount of work to be done in parallel varies linearly with the number of processors. This is more realistic as increasing the number of processors increase the time required to orchestrate the parallel tasks.

Parallel Computing
Fundamentals

- **Serial-parallel scale**

A process can range from "inherently serial" to "embarrasingly parallel":

- An **inherently serial process** is one which functions inside it cannot be parallelize at all, for example, if a function depends on the output of another function before it could begin.
- An **embarrassingly parallel process** is one where there is no inter-dependency between functions executed in the process.
- In the middle of the scale, a **partial parallelizable process** contains functions which can be processed in parallel to a certain extent. For example, if two functions can begin simultaneously, but one must wait for the result of the other to finish.

Fundamentals

Following with the previous characterization and depends on how much the subtasks communicate with/depends on each other, parallelism can be classified as:

- **Fine-grained parallelism**: relatively small amounts of parallel computational work are done between communication events.
- **Coarse-grained parallelism**: relatively large amounts of parallel computational work are done between communication events.
- **Embarrasing parallelism**: subtasks rarely or never communicate. This case is where the **highest parallel speedup** occurs.

If the functions which must be processed are very fast, **running them all parallel may be slower than running all serial**, due to the increase in computational charge associated to the parallelization (communication between threads/cores, waiting for resource availability, etc.).

Fundamentals

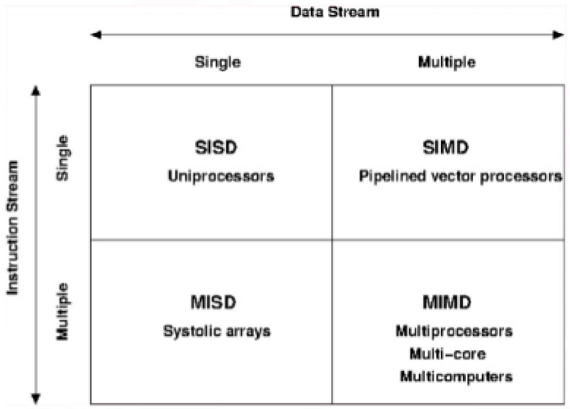
Basic tips

These are given as examples of characteristics of parallelizable processes:

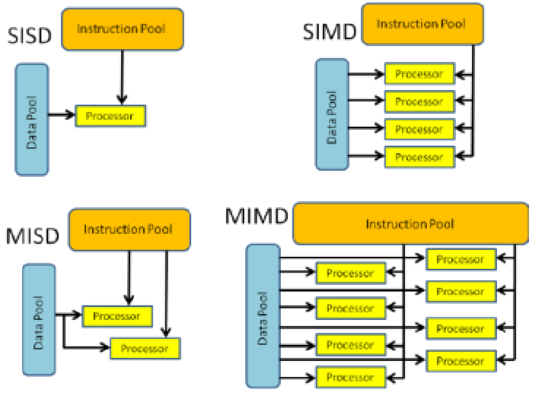
- Tuning models by repeated re-fitting models (for example, grid tuning)
- Apply a transformation to many different variables/tables
- Estimating model quality through cross-validation, bootstrap or other repeated sampling techniques.
- A code bottleneck which can be done in several steps.

Parallel Computing
Fundamentals

A classification of parallel computers is given by the Flynn's taxonomy:

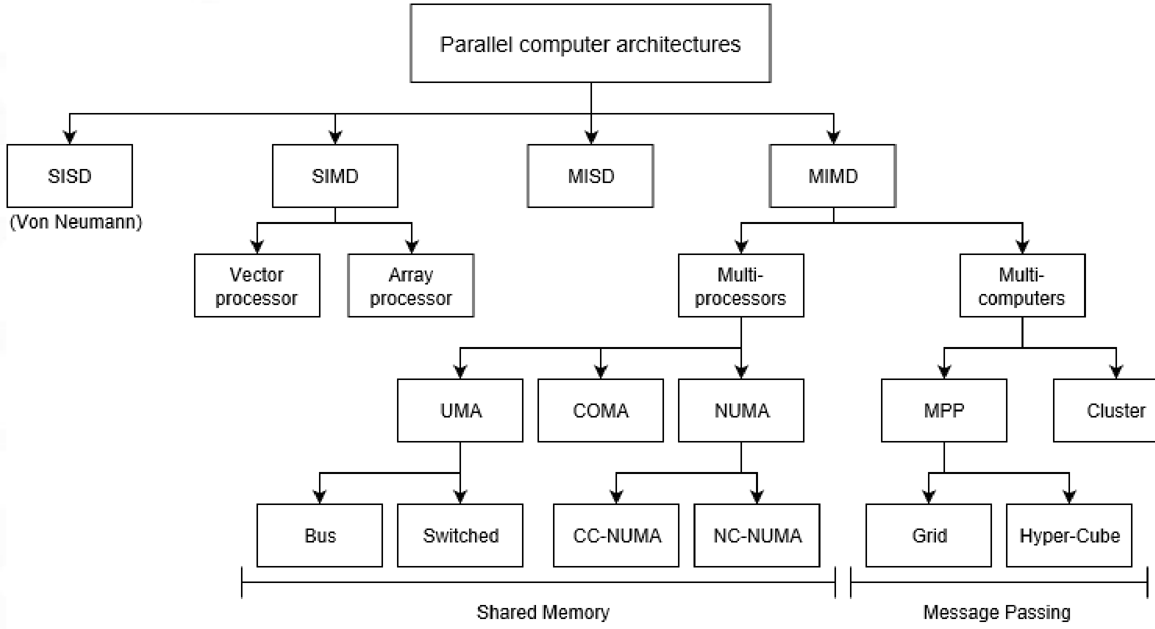


Instruction Stream \ Data Stream	Single	Multiple
Single	**SISD** Uniprocessors	**SIMD** Pipelined vector processors
Multiple	**MISD** Systolic arrays	**MIMD** Multiprocessors, Multi-core, Multicomputers



Parallel Computing
Fundamentals

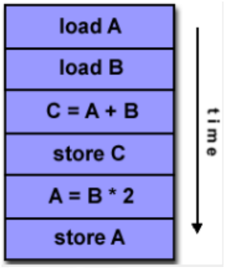
An expansion of the Flynn's taxonomy can be seen in the next figure:



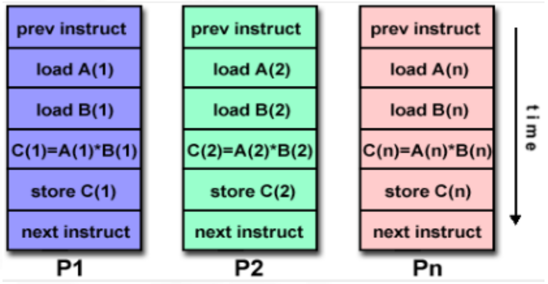
Parallel Computing

Fundamentals

- Single Instruction, Single Data (SISD)

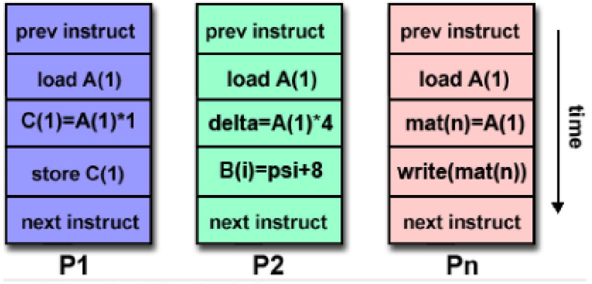


- Single Instruction, Multiple Data (SIMD)

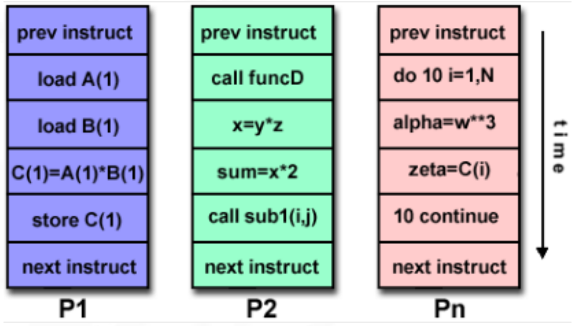


Parallel Computing
Fundamentals

- Multiple Instruction, Single Data (MISD)

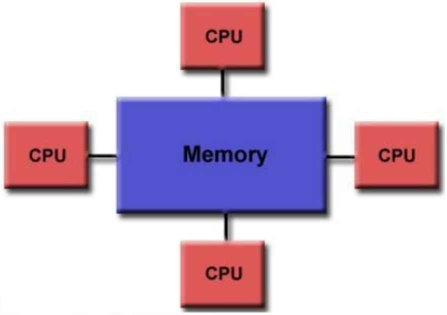


- Multiple Instruction, Multiple Data (MIMD)

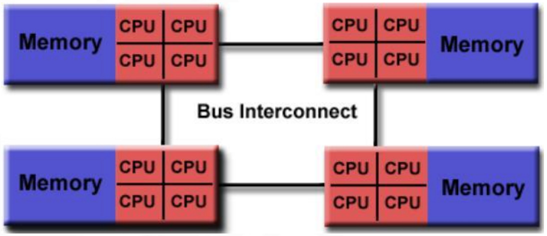


Parallel Computing
Fundamentals

- MIMD Multiprocessor – UMA



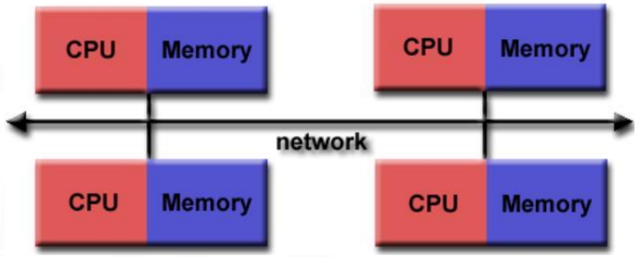
- MIMD Multiprocessor - NUMA



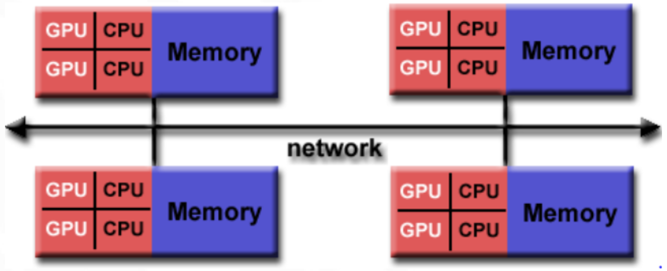
Parallel Computing

Fundamentals

- MIMD Multicomputer – Distributed memory



- MIMD Multicomputer – Shared memory (Hybrid)



Parallel Computing
Fundamentals

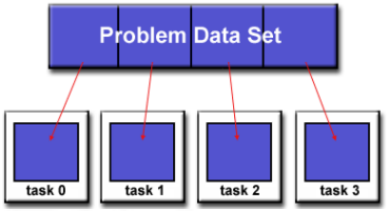
- MIMD Multicomputer – Massive Parallel Processing

Designing Parallel Programs

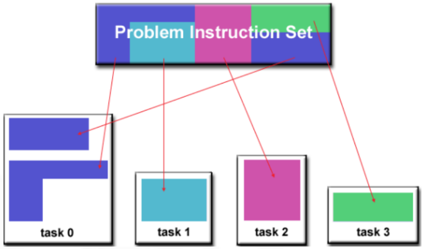
Partitioning

How to break the program into discrete chunks to be distributed:

- Domain decomposition



- Functional decomposition



Designing Parallel Programs

Communications

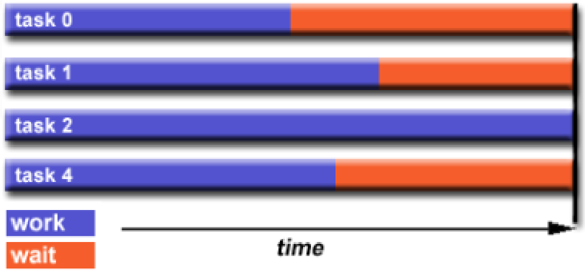
Factors to consider:

- **Communication overhead**: part of the system orchestrates communications instead of "productive work".
- **Latency vs Bandwidth**: sending small messages can cause latency to dominate communications overheads.
- **Visibility of communication**:
  - **Message Passing Model**: communications are explicit and generally quite visible under the control of the programmer.
  - **Data Parallel Model:** communications are invisible to the programmer, particularly on distributed memory architectures.
- **Synchronous vs Asynchronous**
- **Communications scope:** Point-to-point or Collective.

Designing Parallel Programs

Load balancing

Work shall be distributed equally among tasks so that all tasks are kept busy all of the time. Slowest task determines the overall performance:



Load balance is achieved by **equally partitioning** the work (work size is fixed and can be divided) or **dynamic work assignment** (work size is variable or non-predictable).

Designing Parallel Programs

Granularity

Qualitative measure of the ratio of computation to communication. Only to see it again:

- Fine-grain parallelism
- Coarse-grain parallelism
- Embarrasing parallelism

Designing Parallel Programs

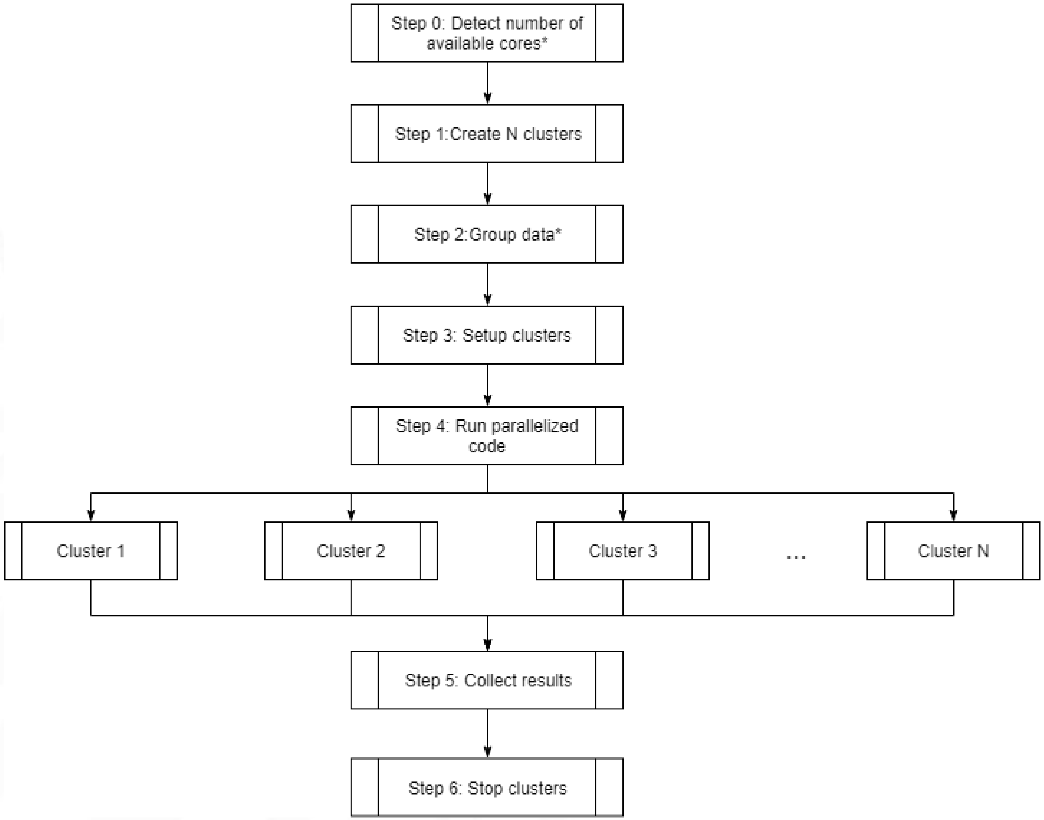
Input/Output

I/O operations are generally regarded as inhibitors to parallelism, because they require order of magnitude more time than memory operations.

Parallel I/O systems are immature or not available for all platforms.

It’s a good practice to confine I/O to specific serial portions of the job and then use parallel communications to distribute data to parallel tasks.

Parallel Computing
Designing Parallel Programs



Parallel Computing

Parallel backends (i.e. clusters)

By default, R will not take advantage of all the cores available on the system. Cores must be made available to R by registering a “parallel backend”, creating a cluster to which computations can be sent.

Packages:

- `doMC`, built on `multicore`, works for unix-alikes systems only.
- `doSNOW`, built on `snow`, works for Windows systems.
- `doParallel`, built on `parallel`, works for both.

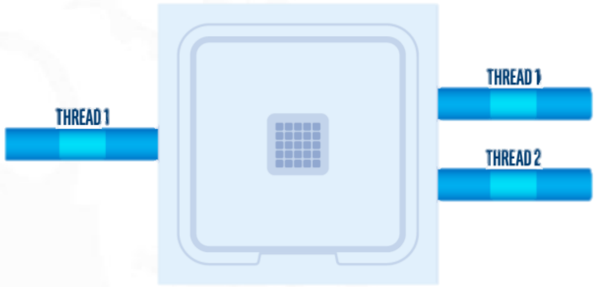
These three packages are used to provide a parallel backend for the `foreach` package, which is the one responsible to process the code in parallel.

Parallel backends (i.e. clusters)

When parallelizing processes, it's always helpful to know the number of CPUs or cores availables.

Function `detectCores()` from the parallel package tries to determine the number of CPU cores in the machine on which R is running. It reports the number of logical cores available.

Logical cores are different from physical cores due to hyper-threading, which allows a thread to be executed in a physical cores in the idle tasks of the rest of threads.

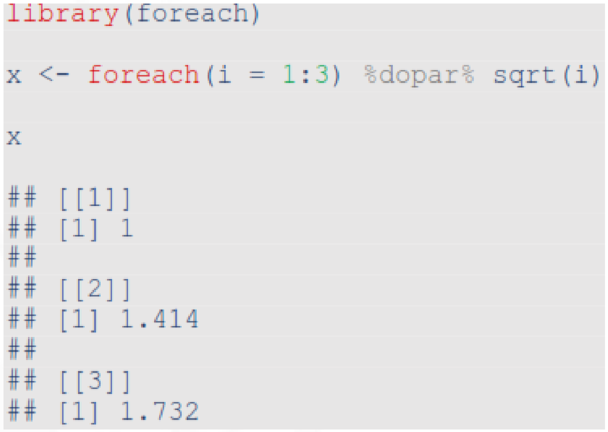


Execution of parallel processes

Simple parallelization

- `foreach`

In this case, there is no need to define the cores/cluster. `foreach` is the parallel analogue to standard `for` loop.

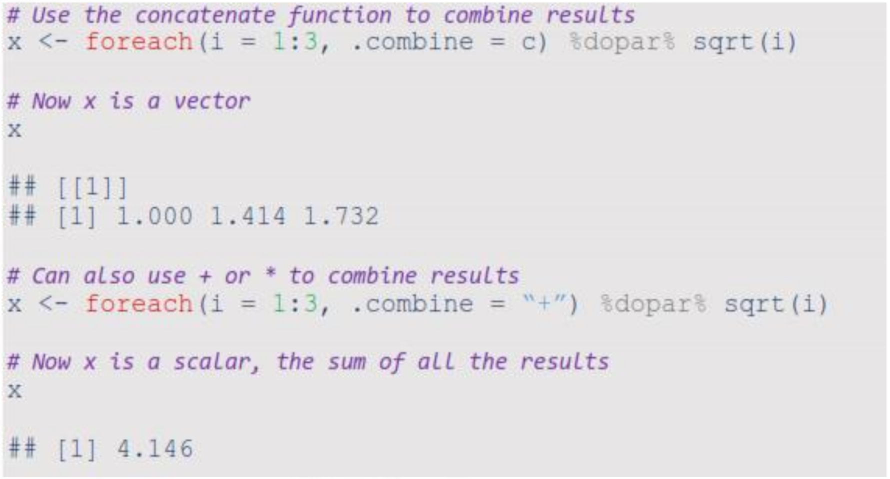


```
library(foreach)

x <- foreach(i = 1:3) %dopar% sqrt(i)

x

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414
##
## [[3]]
## [1] 1.732
```

Parallel Computing

Execution of parallel processes

The combine argument change how the results are reported by the `foreach` function.



```
# Use the concatenate function to combine results
x <- foreach(i = 1:3, .combine = c) %dopar% sqrt(i)

# Now x is a vector
x

## [[1]]
## [1] 1.000 1.414 1.732

# Can also use + or * to combine results
x <- foreach(i = 1:3, .combine = "+") %dopar% sqrt(i)

# Now x is a scalar, the sum of all the results
x

## [1] 4.146
```

Execution of parallel processes

- `parLapply`

The `parallel` package also provides a parallel analogues of the `apply` function.

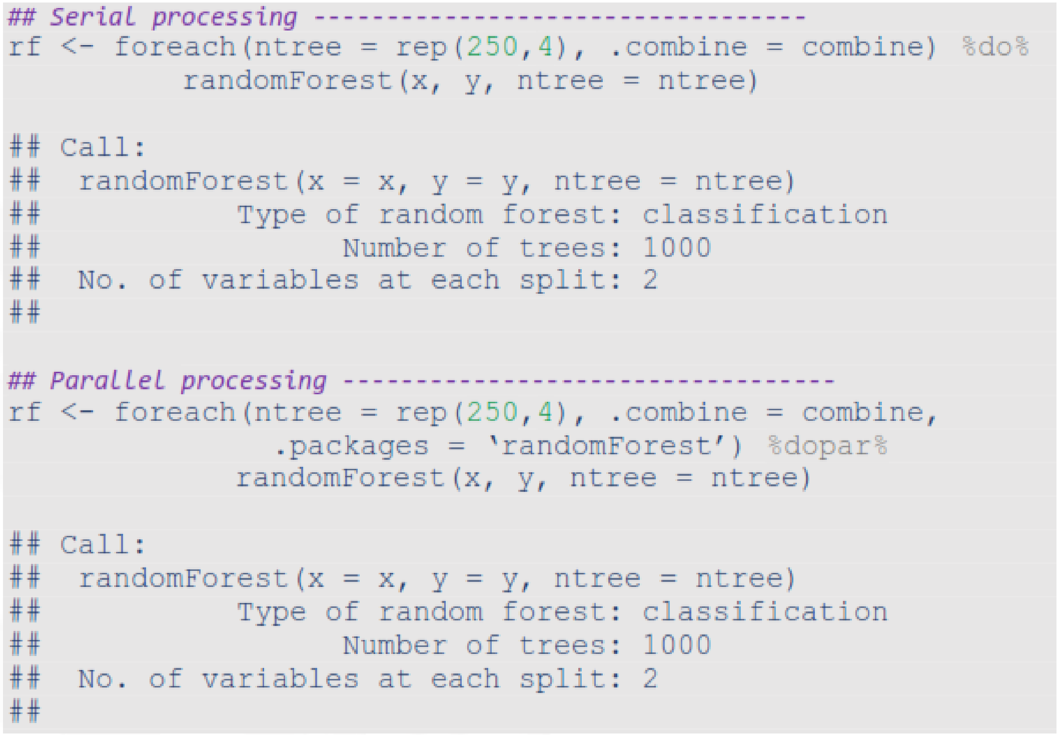
```
library(doParallel)

cl <- makeCluster(3) #Make 3 cluster available in the system

x <- parLapply(cl, list(1,2,3), sqrt)

x

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414
##
## [[3]]
## [1] 1.732
```

If the `xapply()` function must be used with variables that are already declared in the workspace, they must be exported to the cluster using the `clusterExport()` function.

Parallel Computing

Execution of parallel processes

- Parallel training



```
## Serial processing ----------------------------------
rf <- foreach(ntree = rep(250,4), .combine = combine) %do%
            randomForest(x, y, ntree = ntree)

## Call:
##  randomForest(x = x, y = y, ntree = ntree)
##                Type of random forest: classification
##                      Number of trees: 1000
##  No. of variables at each split: 2
##

## Parallel processing ----------------------------------
rf <- foreach(ntree = rep(250,4), .combine = combine,
                  .packages = 'randomForest') %dopar%
               randomForest(x, y, ntree = ntree)

## Call:
##  randomForest(x = x, y = y, ntree = ntree)
##                Type of random forest: classification
##                      Number of trees: 1000
##  No. of variables at each split: 2
##
```

Parallel Computing

Execution of parallel processes

Useful functions:

- `detectCores()`: detect number of logical cores available in the system. Example: `N <- detectCores() - 1`
- `makeCluster(N)`: make N clusters available to use in R. N should be less than the physical cores available in the system in order to not saturate the system. The clusters would communicate using sockets. It returns a cluster object. Example: `cl <- makeCluster(N)`
- `stopCluster(cl)`: make `cl` clusters unavailable to use in R. Example: `stopCluster(cl)`
- `clusterExport(cl, data_name)`: export the variable whose name is data_name ("data_name" must be a character) to the clusters being used. Example: `clusterExport(cl, "dat")`
- `registerDoParallel(cl)`: register the parallel backend with the foreach package. This is needed for most of the functions. Example: `registerDoParallel(cl)`

Parallel Computing

Execution of parallel processes

- `getDoParWorkers()`: returns the number of execution workers there are in the currently registered doPar backend. It can be useful when determining how to split up the work to be executed in parallel. Example: `doPar_workers <- getDoParWorkers()`
- `getDoParVersion()`: returns the version of the currently registered doPar backend. A NULL is returned if no backend is registered. It can be useful to assure that all the cluster are working under the same R version. Example: `doPar_version <- getDoParVersion()`
- `clusterEvalQ()`: evaluates a literal expression on each cluster node. It is useful to load libraries in each of the clusters. Example: `clusterEvalQ(cl, library(caret))`

Parallel Computing

Terminology

There are some definitions that are basic to understand parallel programming:

- **Parallel computer**: multiple processor computer capable of parallel processing.
- **Parallel processing**: information processing with emphasis on the **concurrent manipulation of tasks** belonging to one or more processors solving a single problem. Ideally, parallel processing makes programs run faster because there are more engines (**CPUs** or **cores**) running it.
- **Super Computer** or **High Performance Computer (HPC)**: general purpose computer capable of solving individual problems at an extremely **high computational speed** compared to other computer built in the same time period.

Terminology

- **Core**: general term for either a single core of the multicore processor of the computer or a single machine in a cluster network.
- **Cluster**: collection of objects capable of hosting cores, either a network or a collection of cores of a computer.
- **Critical path**: computing time of the longest chain of dependent calculations.
- **Task**: is a logically discrete section of computational work that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.
- **Shared memory**: model where parallel tasks have the same “picture” of memory and can directly address and access the same logical memory location regardless of where the physical memory exists.

Terminology

- **Distributed memory**: in hardware refers to network-based memory access for physical memory.
- **Communications**: refer to data exchange between workers.
- **Synchronization**: coordination of parallel tasks in real time. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Usually involves waiting by at least one task and can therefore cause a parallel application's wall clock execution time to increase.
- **Granularity**: qualitative measure of the ratio of computation to communication.
  - **Coarse**: relatively large amounts of parallel computational work are done between communication events.
  - **Fine**: relatively small amounts of parallel computational work are done between communication events.

Terminology

- **Parallel overhead:** amount of time required to coordinate parallel tasks, as opposed to doing useful work. This includes:
  - Task start-up time
  - Synchronizations
  - Data communications
  - Software overhead inherent to the system
  - Task termination time
- **Massively parallel:** refers to the hardware that comprises a given parallel system (multiple processor).
- **Scalability**: refers to a parallel system's ability to demonstrate a proportionate increase in speed up with the addition of more resources. Factors that contribute to scalability are:
  - Hardware - particularly memory-CPU bandwidths and network communication properties
  - Application algorithm
  - Parallel overhead related
  - Characteristics of your specific application

Parallel Computing

Terminology

- **Throughput**: number of results a device produces per unit of time.
- **Pipeline**: computation process designed in a consecutive number of independent steps called segments or stages.
- **Sequential vs Parallelism Speedup (s(n))**: ratio between the time needed for the most efficient sequential algorithm (**Ts**) and the time needed for an algorithm incorporating pipelining and/or parallelism (**Tp**):

$$S(n) = \frac{T_s}{T_p}$$

- **Parallelization Speedup (s(n))**: ratio between the time required to run a program on a single processor (**n**=1) and the time required the same program on **n** parallel clusters/cores:

$$S(n) = \frac{T(1)}{T(n)}$$

Parallel Computing

Terminology

- **Degree of parallelism**: total number of processors required to execute a program.
- **Hardware parallelism** vs **Software parallelism**: hardware parallelism is built into the **machine architecture/chip design** (for example, VHDL) and software parallelism is the **concurrent execution of machine language instructions** in a program.
- **Clock rate**: inverse of the constant cycled time, usually measured in nanoseconds, that drives the Central Processing Unit (CPU).
- **Instruction Count (IC)**: number of machine instructions to be executed by a program.
- **Cycles Per Instruction (CPI)**: ratio between the CPU clock cycles for executing a program and the Instruction Count of the same program.

$$CPU\ time\ (T) = \frac{IC \cdot CPI}{Clock\ rate}$$

Terminology

- **Central Processing Unit (CPU)**: electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.
- **Graphics Processing Unit (GPU)**: specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. Their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.
- **Multi-Core Processor**: Single computing component with two or more independent actual CPUs (called “cores”). Multiple cores can run multiple instructions at the same time, increasing overall speed for programs suitable for parallel computing. Cores share the main memory and peripherals, in order to simultaneously process programs.

Terminology

- **Coprocessor**: Computer processor used to supplement the functions of the CPU. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate system performance. A coprocessor may not be a general purpose processor. The coprocessor requires the CPU to fetch the coprocessor instructions and handle all other operations aside from the coprocessor functions.
- **Microprocessor**: multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output. It is an example of sequential digital logic, as it has internal memory.

Bibliography

- Blaise, B. (2018). *Introduction to Parallel Computing*. URL: https://computing.llnl.gov/tutorials/parallel_comp/
- Leach, C. (2014). *Introduction to parallel computing in R*. URL: http://michaeljkoontz.weebly.com/uploads/1/9/9/4/19940979/parallel.pdf
- Mount, J. (2016). *A gentle introduction to parallel computing in R*. URL: http://blog.revolutionanalytics.com/2016/01/a-gentle-introduction-to-parallel-computing-inr.html
- Weston, S. and Calaway, R. (2017). *Getting Started with doParallel and foreach.* URL: https://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf
- Ostrouchov, G., Chen, W. C., Schmidt, D. and Patel, P. (2012). *Programming with Big Data in R*. URL: http://r-pbd.org/
- Kulkarni, V. (2020). *Quantum Parallelism — Where Quantum Computers Get Their Mojo From*. URL: https://towardsdatascience.com/quantum-parallelism-where-quantum-computers-get-their-mojo-from-66c93bd09855

Bibliography

- Böhringer, S. (2013). *Dynamic Parallelization of R Functions*. URL: https://journal.r-project.org/archive/2013/RJ-2013-029/RJ-2013-029.pdf
- Eddelbuettel, D., (2018). *High-Performance and Parallel computing in R.* URL: http://cran.rproject.org/web/views/HighPerformanceComputing.html
- Chen, W. C. and Ostrouchov, G. (2011). URL: https://snoweye.github.io/hpsc/index.html