A group of cubes connected together

Description automatically generated

Task:02

Domain: Algorithm and Data-structure

Mehjabin Mostafa | Real-Time Processing with Divide and Conquer Algorithms in Big Data Applications

**Recursion**

**Definition:**

Recursion is a technique in which a function calls itself repeatedly to find the desired value.There are 3 parts in a recursive occurrence.

* Recursive function
* Base condition
* Recursive Case

In a recursive function base condition and recursive case are used. At first it performs logic that call itself again to perform the same logic but in a smaller input than earlier. And after finding the base case it auto-terminates the whole process.

Base case basically stops recursion and returns the desired value. the function will call itself until finding the base case.

Recursive function presents itself through recursive case. It can contain multiple calls or parameters for the base case to satisfy it and terminate the recursion.

Example:

**find fibonacci series using recursion:**

*#include <iostream>*

*using namespace std;*

*int fib(int n) //Recursive function*

*{*

*if (n == 0) //Base case*

*return 0;*

*if (n == 1 || n == 2) //Base case*

*return 1;*

*else*

*return (fib(n - 1) + fib(n - 2)); // Recursive case*

*}*

*int main()*

*{*

*//calling the function*

*int n = 5;*

*cout << "Fibonacci series of 5 numbers is: ";*

*for (int i = 0; i < n; i++) {*

*cout << fib(i) << " ";*

*}*

*return 0;*

*}*

**Final structure for memorization:**

*#include <iostream>*

*using namespace std;*

*// Generic recursive function*

*returnType functionName(parameters) {*

*// Base Case: Stops the recursion*

*if (baseCondition) {*

*return baseValue;*

*}*

*// Recursive Case: Calls itself with a smaller/simpler input*

*return someCalculation + functionName(smallerInput);*

*}*

*int main() {*

*// Example of calling the recursive function*

*inputType input = someValue;*

*cout << "Result: " << functionName(input) << endl;*

*return 0;*

*}*

**Application of Recursion in Big Data:**

* Hierarchical Data Processing/ Clustering
* Divide and Conquer
* Graph and Tree Processing
* Data Aggregation
* Sorting and Searching

For larger data set recursion should be avoided because stack overflow can happen due to memory usage. Distributive processing like Hadoop or MapReduce can be used instead of recursion for this memory problem. Or iteration algorithm can be implemented.

**Back Tracking:**

Back tracking is a technique that explores different paths to find the best solution and undo the last decision if it reaches a dead end (invalid or incomplete solution). Backtracks always use recursion to explore all the possibilities in a systematic way. From the findings, a outcome will lead to an initial solution and after that if it finds any invalid outcome it will come back to its initial solution to again explore other multiple possibilities.

Backtracking always needs recursion, but recursion doesn’t need backtracking all the time.

**Pseudo code:**

*f (valid* ***solution****):*

*store the* ***solution***

*Return*

*for (all* ***choice****):*

*if (valid* ***choice****):*

***APPLY*** *(****choice****)*

***FIND\_SOLUTIONS*** *(parameters)*

***BACKTRACK*** *(remove* ***choice****)*

*Return*

Example:

*//permutation of given string*

*#include <bits/stdc++.h>*

*using namespace std;*

*void permuteRec(string& s, int idx)*

*{*

*// Base case*

*if (idx == s.size() - 1) {*

*cout << s << endl;*

*return;*

*}*

*for (int i = idx; i < s.size(); i++) {*

*swap(s[idx], s[i]);*

*permuteRec(s, idx + 1);*

*// Backtrack*

***swap(s[idx], s[i]);***

*}*

*}void permute(string& s) {*

*permuteRec(s, 0);*

*}*

*int main(){*

*string s = "ABC";*

*permute(s);*

*return 0;}*

Graph coloring example:

*#include <iostream>*

*#include <vector>*

*using namespace std;*

*// Function to check if the current color assignment is valid*

*bool isSafe(int node, vector<vector<int>>& graph, vector<int>& color, int col) {*

*for (int neighbor = 0; neighbor < graph.size(); neighbor++) {*

*if (graph[node][neighbor] && color[neighbor] == col) {*

*return false;*

*}*

*}*

*return true;*

*}*

*bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>& color, int node) {*

*if (node == graph.size()) {*

*return true; // All vertices are successfully colored*

*}*

*for (int col = 1; col <= m; col++) {*

*if (isSafe(node, graph, color, col)) {*

*color[node] = col;*

*if (graphColoringUtil(graph, m, color, node + 1)) {*

*return true;*

*}*

*color[node] = 0; // Backtrack*

*}*

*}*

*return false;*

*}*

*void graphColoring(vector<vector<int>>& graph, int m) {*

*vector<int> color(graph.size(), 0);*

*if (graphColoringUtil(graph, m, color, 0)) {*

*cout << "Solution exists with " << m << " colors.\n";*

*for (int i = 0; i < color.size(); i++) {*

*cout << "Vertex " << i << " ---> Color " << color[i] << "\n";*

*}*

*} else {*

*cout << "No solution exists with " << m << " colors.\n";*

*}*

*}*

*int main() {*

*vector<vector<int>> graph = {*

*{0, 1, 1, 1},*

*{1, 0, 1, 0},*

*{1, 1, 0, 1},*

*{1, 0, 1, 0}*

*};*

*int m = 3;*

*graphColoring(graph, m);*

*return 0;*

*}*

Job Scheduling problem(pseudo code):

*function schedule(job, time\_slot):*

*if all jobs are scheduled:*

*return true*

*for each machine in available\_machines:*

*if machine can process job at time\_slot:*

*assign job to machine*

*if schedule(next\_job, next\_time\_slot):*

*return true*

*unassign job (backtrack)*

*return false*

Real Life Use of Back-Tracking:

* Puzzle solving/Maze solving
* Network and security
* Pattern Matching
* DNA Sequence alignment
* Protein Structure prediction

For big databack tracking can be a good choice but due to its computational operations some optimization may need.

**Determinist Polynomial VS Non-Deterministic Polynomial**

**Definition (P VS NP):**

Polynomial time problems are those that can be solved by an algorithm and the time required to solve the problem depends on the size of its input. If the time it takes to finish the task grows at a predictable rate (like proportional to a polynomial function of the input size such as n, ), it’s considered to be solvable and tractable(the problem which can be solved in theory as well as practice) in polynomial time. These problems are easily and efficiently solvable and their solutions typically require manageable computational resources, even as the input size grows.

Non polynomial problems are the collection of decision-based problems. The problems are those which cannot be solved efficiently (like in polynomial time) by any normal algorithm because as the input size grows, the time required to solve these problems increases extremely fast, often with complexities like O (), O () etc. It can be solved by non-deterministic machines in polynomial time. It’s a bit hard to solve any problem from scratch but easy to verify if the solution is given.

**Why is it important in Big Data?**

* For efficiency and scalability. Because both P and NP can work with large set of inputs.
* For optimizing problems because without proper optimization finding the best route to solution is often impossible.
* For data analysis and pattern recognition because some NP hard problems require Machine Learning frequent pattern mining and graph problems.

**Real Life examples:**

**Job scheduling problems:**

We can solve this NP hard problems by many algorithms and techniques

* For Subproblems

1. Integer Linear Programming
2. Brach and Bound

* Quick but not Guaranteed Optimal (heuristic Algorithm)

1. Greedy Algorithm
2. Priority Scheduling as adding weights based on importance
3. Dispatching rule

* Complex and Large Instances (Metaheuristic Algorithm)

1. Genetic Algorithm
2. Simulated Annealing
3. Ant colony optimization
4. Tabu Search
5. Particles Swarm Optimization

* Constraint Programming

1. Google OR tools

* Divide and Conquer
* Parallel Processing

1. MapReduce

**Travelling Salesman Problem:**

We can also solve the problem according to the previously stated solutions. Or we can also use hybrid approaches to find a set of solutions.

**Recursive Algorithm Design:**

It’s a method where it divides the whole problem into sub problems, solve recursively the sub problems and combine the result to find the whole solution.

* Base case: A condition that stops the recursion which is the simplest solution of the problem.
* Recursive Case: Logic that takes less input so that it can create smaller sub problems.
* Self-referential which will solve the sub problem by calling itself.
* Ensure termination to avoid infinite recursion.

Example:

Recursive Directory Size Calculation for Large Inputs:

*#include <iostream>*

*#include <vector>*

*#include <string>*

*using namespace std;*

*struct File {*

*string name;*

*bool isDirectory;*

*int size; //(valid only if isDirectory == false)*

*vector<File\*> children;*

*};*

*int calculateTotalSize(File\* node) {*

*if (!node->isDirectory) {*

*return node->size; // Base case*

*}*

*int totalSize = 0;*

*for (File\* child : node->children) {*

*totalSize += calculateTotalSize(child); // Recursive case*

*}*

*return totalSize;*

*}*

*//function*

*File\* createDirectory() {*

*cout << "Enter name of the directory/file: ";*

*string name;*

*cin >> name;*

*cout << "Is this a directory? (1 for Yes, 0 for No): ";*

*bool isDirectory;*

*cin >> isDirectory;*

*File\* node = new File{name, isDirectory, 0, {}};*

*if (!isDirectory) {*

*cout << "Enter size of the file: ";*

*cin >> node->size;*

*} else {*

*int numChildren;*

*cout << "Enter the number of files/subdirectories in " << name << ": ";*

*cin >> numChildren;*

*for (int i = 0; i < numChildren; i++) {*

*cout << "Adding child " << i + 1 << " to directory " << name << endl;*

*node->children.push\_back(createDirectory());*

*}*

*}*

*return node;*

*}*

*int main() {*

*cout << "Creating root directory..." << endl;*

*File\* root = createDirectory();*

*cout << "Calculating total size of the directory structure..." << endl;*

*int totalSize = calculateTotalSize(root);*

*cout << "Total size: " << totalSize << " bytes" << endl;*

*return 0;*

*}*

In Recursive algorithm design, time complexity will remain the same theoretically for larger inputs, but actual runtime increases due to more operations performed with larger datasets.

Space complexity grows larger as the inputs grows because the memory used by the recursion stack increases. To solve this problem tail recursion or converting recursive algorithms to iterative algorithms can reduce space complexity, as they eliminate the recursion stack.

**Real Life application with big set of Data:**

* Graph Traversal
* Decision Tree
* Hierarchical data processing
* Big sets of data sorting

**Basics of Parallel Processing:**

Parallel Processing means simultaneous implementation of multiple tasks which are divided into smaller parts as subproblems and running them at the same time on multiple processors or computers.

Parallel processing divides a task between two or more microprocessors. Typically, a complex task is divided into multiple parts using a specialized software tool that assigns each part to a processor based on the task's component elements. Larger tasks are broken into multiple smaller parts that are appropriate for the number, type and size of available processing units. Then, each processor completes its part, and the software tool reassembles the data and executes the task.

**Types Based on Granularity**:

Parallel processes are either **fine-grained** or **coarse-grained**. In fine-grained parallel processing, tasks communicate with one another multiple times. This is suitable for processes that require real-time data.

Coarse-grained parallel processing deals with larger pieces of a task and requires less frequent communication between processors. This type of processing is more useful in applications that have tasks with minimal interdependence and can be naturally divided into larger parts.

**Types Based on Memory Access:**

Shared Memory Parallelism:

* Multiple processors share a single memory space. Share the same I/O bus or data path and a single copy of the OS overseas with all the processors.

Distributed Memory Parallelism:

* This technique uses many processors that work on different parts of the task and each processor has its own memory, and communication occurs via a network.

**Types Based on Architecture:**

SISD (Single Instruction, Single Data):

Traditional sequential processing (not parallel).

SIMD (Single Instruction, Multiple Data):

Executes the same instruction on multiple data points simultaneously.

MISD (Multiple Instructions, Single Data):

Multiple instructions operate on the same data stream. It is rarely used in practice but theoretically valuable for specialized tasks.

.

MIMD (Multiple Instructions, Multiple Data):

Multiple processors execute different instructions on different data simultaneously.

**Why is it important in big data:**

* Faster data processing
* Cost efficiency
* Fault tolerance
* Resource Optimization

**Usage:**

OpenMP (open multi-processing):

It’s a parallel programming model for multi-threaded applications. It supports multi-platform shared memory multiprocessing programming that can be done by C, C++ and Fortan. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. Threads exist within the resources of a *single* process. Without the process, they cease to exist.

Typically, the number of threads match the number of machines.

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

parallelization can be as simple as taking a serial program and inserting compiler directives....

in general, this is way more complex

OpenMP uses the fork-join model of parallel execution

**FORK**: the master thread then creates a team of parallel threads.

The statements in the program that are enclosed by the parallel region construction are then executed in parallel among the various team threads.

**JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

A diagram of a task

Description automatically generated

Example (matrix addition with OpenMp):

*#include <iostream>*

*#include <omp.h>*

*#define N 1000*

*Namespace std;*

*int main() {*

*int A[N][N], B[N][N], C[N][N];*

*for (int i = 0; i < N; i++)*

*for (int j = 0; j < N; j++) {*

*A[i][j] = i + j;*

*B[i][j] = i - j;*

*}*

*#pragma omp parallel for collapse(2)(direction for the compiler to take specific action regarding the next two nested loops as a single loop for parallelization.)*

*for (int i = 0; i < N; i++) {*

*for (int j = 0; j < N; j++) {*

*C[i][j] = A[i][j] + B[i][j];*

*}*

*}*

*cout << "Matrix addition completed!" << endl;*

*return 0;*

*}*

**MPI (message passing interface):**

It is a message passing library standards based on the consensus of the MPI forum. It offers a portable, efficient and flexible standard for message passing that will be used for writing message passing programs. It has become the industry standard for writing message passing programs on HPC platforms. In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor.

MPI implementors adapted their libraries to handle both distributed and shared type of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.

A diagram of a network

Description automatically generated

Example:

*#include <mpi.h>*

*#include <iostream>*

*#define N 1000*

*Namespace std;*

*int main(int argc, char\*\* argv) {*

*int rank, size;*

*int A[N], B[N], C[N];*

*MPI\_Init(&argc, &argv);*

*MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);*

*MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);*

*int chunk = N / size;*

*if (rank == 0) {*

*for (int i = 0; i < N; i++) {*

*A[i] = i;*

*B[i] = 2 \* i;*

*}*

*}*

*int local\_A[chunk], local\_B[chunk], local\_C[chunk];*

*MPI\_Scatter(A, chunk, MPI\_INT, local\_A, chunk, MPI\_INT, 0, MPI\_COMM\_WORLD);*

*MPI\_Scatter(B, chunk, MPI\_INT, local\_B, chunk, MPI\_INT, 0, MPI\_COMM\_WORLD);*

*for (int i = 0; i < chunk; i++) {*

*local\_C[i] = local\_A[i] + local\_B[i];*

*}*

*MPI\_Gather(local\_C, chunk, MPI\_INT, C, chunk, MPI\_INT, 0, MPI\_COMM\_WORLD);*

*if (rank == 0) {*

*cout << "Matrix addition completed!" << endl;*

*}*

*MPI\_Finalize();*

*return 0;*

*}*

Overall OpenMP is good for multicore machines such as personal computers or servers and MPI is for large-scale distributed systems like supercomputers or cloud environments.

A diagram of a computer network

Description automatically generated

**Hadoop Framework:**

Apache Hadoop is a powerful framework that enables the distributed processing of large datasets across clusters of computers. It is designed to handle big data and is based on the MapReduce programming model, which allows for the parallel processing of large datasets.

****

Fig: Clustered Computer

It can process large scale data on clusters parallel to hundreds of operations. Hadoop processes large datasets across distributed clusters using HDFS to distribute data and MapReduce for parallel processing. It optimizes tasks with data locality, manages resources via YARN, and ensures scalability and fault tolerance through automatic task redistribution among nodes, maximizing efficiency and reliability in data handling.

**Features of hadoop:**

* It is fault tolerance.
* It is highly available.
* its programming is easy.
* It has huge flexible storage.
* It is low cost.

MapReduce is the computational model that Hadoop uses to process large datasets in parallel. It divides a job into smaller tasks that can be executed concurrently across multiple nodes. The Hadoop can run the MapReduce program written in various languages such as Java, Ruby, and Python. One of the beneficial factors that MapReduce aids is that MapReduce programs are inherently parallel, making the very large scale easier for data analysis. The model consists of two main phases:

* Map Phase

In the Map phase, input data is split into chunks and processed independently by Mapper tasks. Each Mapper reads a block of data, processes it, and produces intermediate key-value pairs.

**Key Characteristics:**

1. Data Splitting: Input data is split into smaller chunks.
2. Independent Processing: Each chunk is processed independently by a Mapper task.
3. Key-Value Pairs: Mappers output intermediate key-value pairs.

* Reduce Phase

n the Reduce phase, the intermediate key-value pairs produced by the Mappers are shuffled and sorted, then processed by Reducer tasks to produce the final output. Each Reducer processes all values associated with a particular key.

**Key Characteristics:**

1. Shuffling and Sorting: Intermediate data is shuffled and sorted based on keys.
2. Aggregating Results: Reducers aggregate and process values for each key.
3. Final Output: Reducers produce the final output, which is written to HDFS.

**Usage in Big Data:**

Hadoop's ability to handle parallel processing of large datasets makes it suitable for various applications, including:

* Data Analytics: Processing and analyzing vast amounts of structured and unstructured data.
* Machine Learning: Training models on large datasets distributed across the cluster.
* ETL Processes: Extracting, transforming, and loading large volumes of data from different sources.
* Log Analysis: Analyzing logs from web servers, applications, and devices to gain insights.

Example of MapReduce(pseudo code):

*map(String input\_key, String input\_value):*

*// input\_key: document name*

*// input\_value: document contents*

*for each word w in input\_value:*

*EmitIntermediate(w, "1");*

*reduce(String output\_key, Iterator intermediate\_values):*

*// output\_key: a word*

*// output\_values: a list of counts*

*int result = 0;*

*for each v in intermediate\_values:*

*result = result + ParseInt(v);*

*Emit(output\_key, AsString(result));*