UNIT VI

Multithreaded Algorithms – Introduction,

Multithreading हा एक तंत्र आहे ज्यामध्ये **एकाच प्रोग्राममध्ये एकापेक्षा जास्त threads** lightweight processes होतात. हे algorithms एकाच वेळी अनेक tasks efficiently execute करण्यासाठी design केलेले असतात.

उदाहरणार्थ:

- **Sorting algorithms** – QuickSort किंवा MergeSort हे multithreading वापरून वेगाने काम करू शकतात.
- **Matrix multiplication** – Threads वेगवेगळ्या matrix parts वर काम करू शकतात.

Multithreading चे फायदे:

1. **Performance Improvement**: अनेक cores असलेल्या system वर एकाच वेळी अनेक tasks parallel चालू राहतात.
2. **Resource Utilization**:
   Process idle न ठेवता CPU resources पूर्णतः वापरल्या जातात.
3. **Scalability**:
   मोठ्या data handling साठी algorithms scalable होतात.

कसा काम करतो?

1. **Thread creation**: प्रत्येक task साठी स्वतंत्र thread तयार केला जातो.
2. **Synchronization**: Threads मध्ये data conflicts टाळण्यासाठी synchronization आवश्यक असतो.

   उदाहरण: Java मध्ये `synchronized` keyword वापरून data safe ठेवला जातो.

3. **Load Balancing**: प्रत्येक thread ला योग्य प्रमाणात काम दिले जाते.

मर्यादा:

- **Deadlock**: दोन threads एकमेकांवर dependent असतील तर process अडकू शकतो.
- **Debugging**: Multithreaded algorithms debugging करणे कठीण असते.
- **Overhead**: Threads निर्माण करणे आणि synchronize करणे यामुळे काहीवेळा performance कमी होतो.

थोडक्यात:

Multithreaded algorithms हे modern computing चे future आहेत जे performance boost आणि scalability साठी उपयुक्त ठरतात. परंतु यासाठी proper design आणि debugging आवश्यक आहे.

Performance measures,

Algorithm किंवा system ची कार्यक्षमता मोजण्यासाठी **performance measures** वापरले जातात. हे measures algorithm किती effectively आणि efficiently काम करते हे ठरवण्यासाठी महत्त्वाचे आहेत.

1. Execution Time (कार्यकाल):

- Algorithm ला काम पूर्ण करायला लागणारा वेळ.
- **उदाहरण**: Sorting algorithm ला 1000 elements sort करायला लागणारा वेळ.
- **Formula**: T(n)T(n)T(n) = Time as a function of input size nnn.
- "कार्य जितक्या कमी वेळेत पूर्ण होईल तितकी algorithm अधिक efficient समजली जाते."

2. Space Complexity (साठवण क्षमता):

- Algorithm कार्यान्वित करताना लागणाऱ्या memory ची गरज.
- Example: MergeSort जास्त memory वापरतो कारण तो additional arrays निर्माण करतो.
- "ज्या algorithm ला कमी memory लागते, ती अधिक चांगली मानली जाते."

3. Throughput (उत्पन्न दर):

- एका निश्चित कालावधीत algorithm किती काम पूर्ण करू शकतो.
- Example: Web server एका सेकंदात किती requests process करू शकतो.
- "Throughput जितका जास्त तितका system चांगला."

4. Latency (प्रतिसाद वेळ):

- Algorithm किंवा system ला एका task साठी सुरूवात आणि पूर्ण होण्यासाठी लागणारा वेळ.
- Example: User input वर search result मिळण्याचा delay.
- "Latency कमी असेल तर user ला result लगेच मिळतो."

5. Scalability (वाढवण्याची क्षमता):

- Input size किंवा system resources वाढवल्यावर algorithm कसा perform करतो.
- Example: Sorting algorithm मोठ्या datasets वर किती चांगला काम करतो.
- "Scalability महत्त्वाची आहे कारण systems आज मोठ्या प्रमाणावर काम करत आहेत."

6. Accuracy (अचूकता):

- Algorithm चे output योग्य आहे का?
- Example: Machine learning model predictions किती accurate आहेत.
- "Output जितके अचूक तितका algorithm जास्त विश्वासार्ह."

7. Energy Efficiency (ऊर्जेचा वापर):

- Algorithm कार्यान्वित करताना लागणारी ऊर्जा.
- "Energy-efficient algorithms server आणि devices साठी उपयुक्त ठरतात."

8. Reliability (विश्वासार्हता):

- Algorithm consistently योग्य output देते का?
- Example: Banking systems मध्ये calculations नेहमी बरोबर असणे आवश्यक आहे.

Analyzing multithreaded algorithms,

Multithreaded algorithms चे performance आणि efficiency analyze करणे हे एक महत्त्वाचे पाऊल आहे, कारण यामुळे algorithm च्या scalability आणि reliability चे परीक्षण करता येते. Analysis करताना खालील factors लक्षात घेतले जातात:

1. Speedup (गतीवाढ):

- Multithreading मुळे sequential algorithm च्या तुलनेत किती वेगाने काम होते.
- **Formula**:

$$\text{Speedup} = \frac{T_{seq}}{T_{par}}$$

जिथे Tseq = Sequential execution time आणि Tpar = Parallel execution time.

- **Example**: Sorting 1000 elements sequentially घेणारा वेळ 10 सेकंद असेल, तर multithreading मुळे तो 2 सेकंदांवर येईल. Speedup = 10/2 = 5.
- "Speedup जितका जास्त, algorithm तितकी efficient."

2. Efficiency (कार्यक्षमता):

- किती threads effectively resources वापरत आहेत.
- **Formula**:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

- "जर efficiency कमी असेल तर thread utilization योग्य नाही."

3. Scalability (वाढीची क्षमता):

- अधिक threads किंवा cores जोडल्यावर performance कसे बदलते.
- **Example**: Sorting large datasets वर 4 threads वापरणे आणि नंतर 8 threads वापरणे यातील performance चा फरक.
- "Scalable algorithm मोठ्या प्रणालींवरही चांगले perform करू शकतो."

4. Amdahl's Law (मर्यादा):

- Multithreading चे limitations सांगणारा महत्त्वाचा principle.
- **Formula**:

$$\text{Speedup} \leq \frac{1}{(1-P) + \frac{P}{N}}$$

  P = Parallelizable fraction, N = Number of threads.

- "जर algorithm मध्ये parallelizable भाग कमी असेल तर threads वाढवूनही performance फारसा सुधारत नाही."

5. Overhead (अतिरिक्त भार):

- Threads तयार करणे, synchronization, आणि communication यासाठी लागणारा वेळ.
- **Example**: जर 4 threads मुळे 2 सेकंदाचा फायदा होत असेल, पण synchronization साठी 1 सेकंद लागत असेल, तर actual फायदा कमी होतो.
- "Overhead कमी ठेवणे गरजेचे आहे."

6. Load Balancing (कामाचे समतोल वाटप):

- सर्व threads ना workload समान प्रमाणात दिले गेले आहे का?
- **Example**: Sorting algorithm मध्ये एक thread ला 100 elements आणि दुसऱ्या thread ला 1000 elements दिल्यास imbalance होईल.
- "Workload balance नसेल तर काही threads idle राहतील, आणि performance कमी होईल."

7. Deadlocks आणि Race Conditions (डेडलॉक आणि स्पर्धा अटी):

- **Deadlock**: दोन threads resources साठी एकमेकांवर depend असल्याने process थांबते.
- **Race Condition**: दोन threads एकाच वेळी shared data access करत असतील, तर unpredictable behavior होतो.
- "Thread synchronization योग्य नसेल तर deadlocks आणि race conditions होऊ शकतात."

8. Memory Usage (स्मृतीचा वापर):

- Multithreaded algorithm मध्ये threads additional memory consume करतात का?
- Example: Recursive algorithms, जसे parallel QuickSort, stack memory जास्त वापरतात.

9. Testing आणि Debugging (चाचणी व डिबगिंग):

- Multithreaded algorithms चा output consistently योग्य येतो का?
- Tools: Debugging साठी tools जसे GDB, Visual Studio Debugger वापरले जातात.

Parallel loops,

Parallel loops म्हणजे **loop मधील iterations एकाच वेळी (concurrently) execute करणे**, ज्यामुळे कार्यक्षमता (performance) वाढते आणि मोठ्या data sets वर operations जलद होतात. हे **multithreading किंवा multiprocessing** वापरून साध्य केले जाते.

Parallel Loops चे मुख्य तत्त्व:

सामान्य loop मध्ये, प्रत्येक iteration sequentially (क्रमाने) execute होते.
Parallel loops मध्ये iterations स्वतंत्र असल्या कारणाने **वेगळ्या threads किंवा processors** वर एकाच वेळी चालतात.

उदा. (सामान्य Loop):

```
java
Copy code
for (int i = 0; i < n; i++) {
    result[i] = process(data[i]);
}
```

उदा. (Parallel Loop):

```
java
Copy code
parallel_for (int i = 0; i < n; i++) {
    result[i] = process(data[i]);
}
```

फायदे (Advantages):

1. **वेगवान (Faster Execution):** मोठ्या data sets process करणे जलद होते.
    - "Parallel loop वापरून computation time लक्षणीय कमी करता येतो."
2. **Efficient Resource Utilization:** उपलब्ध cores आणि threads effectively वापरले जातात.
3. **Scalability:** मोठ्या computational tasks साठी उपयुक्त.

मर्यादा (Limitations):

1. **Dependencies (अवलंबन):**
    - जर loop मधील iterations एकमेकांवर अवलंबून असतील, तर parallelization शक्य नाही.
    - "Data dependency असल्यास parallel loop काम करत नाही."

2. **Overhead:**
    - Threads तयार करणे आणि synchronize करणे यामुळे काही वेळा performance कमी होतो.
3. **Debugging:**
    - Parallel loops चा debugging process कठीण असतो.

वापर कधी करावा? (When to Use):

1. **Independent Iterations:** प्रत्येक iteration इतरांवर अवलंबून नसताना.
    - Example: Array processing, image pixel manipulation.
2. **Large Data Sets:** मोठ्या प्रमाणावर data वर computation करताना.
3. **Hardware Utilization:** Multicore processors असतील तर.

तंत्रज्ञान:

Parallel loops साठी काही libraries आणि frameworks:

- **Java:** `ForkJoinPool`, `Parallel Streams`.
- **Python:** `joblib`, `multiprocessing`.
- **C++:** `OpenMP`.
- **MATLAB:** `parfor`.

Race conditions.

Race condition ही एक समस्या आहे जी multithreading किंवा multiprocessing मध्ये होते, जेव्हा **एकाच resource वर दोन किंवा अधिक threads एकाच वेळी access करण्याचा प्रयत्न करतात**, आणि त्याचा output unpredictable होतो.

Race Condition कधी घडते?

- **Shared Resources (सामायिक संसाधनं)**: जेव्हा दोन किंवा अधिक threads/shared memory/data structure वापरतात.
  उदाहरण: दोन threads एका variable ची value update करत असतील.
- **No Synchronization (समकालीनतेचा अभाव)**:
  जर threads च्या operations साठी coordination नसेल, तर race condition निर्माण होऊ शकते.
- "Synchronization नसल्यामुळे data inconsistent होऊ शकतो."

उदाहरण (Example):

Code Example (English):

```
java
Copy code
class Counter {
    int count = 0;
```

```
    void increment() {
        count++;
    }
}

Counter counter = new Counter();

// Two threads incrementing the same counter
Thread t1 = new Thread(counter::increment);
Thread t2 = new Thread(counter::increment);

t1.start();
t2.start();
```

Output:

- Output कधी 1, कधी 2 येऊ शकतो, कारण दोन threads एकाच वेळी `count++` execute करू शकतात.

"Variable `count` च्या update process मध्ये अर्धवट changes होत असल्यामुळे inconsistent result मिळतो."

Race Condition चे परिणाम:

1. **Data Corruption (डेटा भ्रष्ट होणे)**:
   Output चुकीचा होतो.
   Example: Bank transaction मध्ये चुकीची value update होणे.
2. **Unpredictable Behavior (अप्रत्याशित वर्तन)**:
   Program expected result देत नाही.

Race Condition टाळण्यासाठी उपाय:

1. **Synchronization (समकालीनता):**
   - Threads चा access नियंत्रित करण्यासाठी synchronization वापरा.
   - **Example**: Java मध्ये `synchronized` block.

     ```
     java
     Copy code
     synchronized(this) {
         count++;
     }
     ```

2. **Locks (कुलूपे):**
   - Thread-safe operations साठी `lock` किंवा `mutex` वापरा.
   - "Locks resource वर एकाच वेळी फक्त एका thread ला access देतात."
3. **Atomic Variables:**
   - Atomic operations race condition टाळतात.
   - Example: Java मध्ये `AtomicInteger` वापरणे.
4. **Avoid Shared State (सामायिक स्थिती टाळा):**
   - Independent variables किंवा message-passing mechanisms वापरा.

Problem Solving using Multithreaded Algorithms – Multithreaded matrix multiplication,

Multithreading वापरून matrix multiplication करणं computational tasks effectively solve करण्याचा एक प्रभावी मार्ग आहे. यामध्ये प्रत्येक thread ला एक विशिष्ट काम assign केलं जातं, ज्यामुळे काम parallel होऊन वेळ वाचतो.

matrix multiplication

Suppose दोन matrices आहेत:

- A(m x n dimension)
- B (n x p dimension)
  Output matrix C (m x p dimension)

Formula:

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$$

पारंपरिक पद्धतीत प्रत्येक element साठी computation sequentially होते, ज्यामुळे वेळ अधिक लागतो.

Multithreading वापरून Matrix Multiplication

1. कसा कार्य करतो?

- प्रत्येक thread ला C मधील काही rows किंवा columns calculate करण्याचं काम दिलं जातं.
- **Example:**
  - जर A आणि B मधे 4x4 ची matrices असतील, तर 4 threads प्रत्येक row independently calculate करू शकतात.

2. Steps for Implementation:

1. **Thread Creation:**
   प्रत्येक thread ला काही rows किंवा columns assign करा.
2. **Parallel Computation:**
   Threads independently त्या assign केलेल्या part साठी calculation करतील.
3. **Synchronization:**
   Output matrix C मध्ये result add करताना threads synchronize केले जातात.

3. उदाहरण – कोड स्केच (Java):

```
java
Copy code
```

```
class MatrixMultiplication extends Thread {
    int row, col;
    int[][] A, B, C;

    MatrixMultiplication(int row, int col, int[][] A, int[][] B, int[][] C)
{
        this.row = row;
        this.col = col;
        this.A = A;
        this.B = B;
        this.C = C;
    }

    public void run() {
        C[row][col] = 0;
        for (int k = 0; k < B.length; k++) {
            C[row][col] += A[row][k] * B[k][col];
        }
    }
}

public class MultithreadedMatrix {
    public static void main(String[] args) {
        int[][] A = { {1, 2}, {3, 4} };
        int[][] B = { {5, 6}, {7, 8} };
        int[][] C = new int[A.length][B[0].length];

        Thread[] threads = new Thread[A.length * B[0].length];
        int count = 0;

        for (int i = 0; i < A.length; i++) {
            for (int j = 0; j < B[0].length; j++) {
                threads[count] = new MatrixMultiplication(i, j, A, B, C);
                threads[count].start();
                count++;
            }
        }

        for (int i = 0; i < count; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Print result
        for (int i = 0; i < C.length; i++) {
            for (int j = 0; j < C[0].length; j++) {
                System.out.print(C[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

4. Multithreading चे फायदे:

- **Speedup:** Threads एकाच वेळी computations करू शकतात.

- **Efficiency:** CPU cores effectively वापरले जातात.

5.Challenge:

- **Synchronization issues:** दोन threads एकाच memory location वर काम करत असतील तर error होऊ शकतो.
- **Overhead:** Threads तयार करण्यासाठी आणि manage करण्यासाठी additional resources लागतात.

Multithreadedmerge sort.

Merge Sort हा divide-and-conquer पद्धतीवर आधारित sorting algorithm आहे. Multithreaded Merge Sort मध्ये **threads** वापरून sorting प्रक्रिया parallelize केली जाते, ज्यामुळे मोठ्या डेटा सेटसाठी sorting वेगाने होते.

Algorithm ची पायरी:

1. **Divide (विभाजन):**
   - Input array दोन sub-arrays मध्ये विभागले जाते.
   - प्रत्येक sub-array साठी independent threads तयार केले जातात.
2. **Sort (क्रमवारी लावा):**
   - प्रत्येक sub-array वर स्वतंत्र merge sort कार्यान्वित करा.
   - जर sub-array खूप मोठा असेल, तर त्यालाही further divide करा आणि नवे threads तयार करा.
3. **Merge (एकत्र करा):**
   - Sub-arrays चे sorted parts एकत्र करून पूर्ण sorted array तयार करा.

Multithreading कसा वापरायचा?

1. **Thread Creation:**
   - प्रत्येक sub-array साठी thread तयार करा.
   - Java मध्ये, `Thread` किंवा `ExecutorService` वापरले जाऊ शकते.
2. **Thread Synchronization:**
   - Sorting आणि merging दरम्यान threads मधील समन्वय राखणे आवश्यक आहे.
   - Shared resources टाळण्यासाठी, thread-safe structures वापराव्यात

Code Example (Java):

```
java
Copy code
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

public class MultithreadedMergeSort {
    static class MergeSortTask extends RecursiveAction {
```

```
        private int[] array;
        private int left, right;

        public MergeSortTask(int[] array, int left, int right) {
            this.array = array;
            this.left = left;
            this.right = right;
        }

        @Override
        protected void compute() {
            if (left < right) {
                int mid = (left + right) / 2;

                MergeSortTask leftTask = new MergeSortTask(array, left,
mid);
                MergeSortTask rightTask = new MergeSortTask(array, mid + 1,
right);

                invokeAll(leftTask, rightTask);

                merge(array, left, mid, right);
            }
        }

        private void merge(int[] array, int left, int mid, int right) {
            int n1 = mid - left + 1;
            int n2 = right - mid;

            int[] L = new int[n1];
            int[] R = new int[n2];

            for (int i = 0; i < n1; i++)
                L[i] = array[left + i];
            for (int j = 0; j < n2; j++)
                R[j] = array[mid + 1 + j];

            int i = 0, j = 0, k = left;
            while (i < n1 && j < n2) {
                if (L[i] <= R[j]) {
                    array[k] = L[i];
                    i++;
                } else {
                    array[k] = R[j];
                    j++;
                }
                k++;
            }

            while (i < n1) {
                array[k] = L[i];
                i++;
                k++;
            }

            while (j < n2) {
                array[k] = R[j];
                j++;
                k++;
            }
        }
```

```
    }

    public static void main(String[] args) {
        int[] array = {38, 27, 43, 3, 9, 82, 10};
        ForkJoinPool pool = new ForkJoinPool();

        MergeSortTask task = new MergeSortTask(array, 0, array.length - 1);
        pool.invoke(task);

        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

Multithreaded Merge Sort चे फायदे:

1. **वेगाने Sorting**: Threads वापरल्यामुळे मोठ्या arrays sorting ची प्रक्रिया जलद होते.
2. **Utilization of Cores**: CPU च्या multiple cores चा पूर्ण उपयोग होतो.
3. **Scalable**: Input size वाढला तरी performance impact कमी होतो.

Limitations:

1. **Thread Overhead**: खूप छोटे arrays साठी threads तयार करणे खर्चिक होऊ शकते.
2. **Synchronization Issues**: Merge process मध्ये threads नीट synchronize करावे लागतात.
3. **Memory Usage**: Merge process साठी अतिरिक्त memory लागते.

Distributed Algorithms –

Introduction,

Distributed Algorithms म्हणजे असे algorithms जे **distributed systems** वर कार्य करतात. Distributed system म्हणजे असा system ज्यामध्ये अनेक interconnected nodes (computers) एकत्र येऊन task पूर्ण करतात.

उदाहरण:

- Banking systems, ज्यामध्ये data अनेक servers वर distributed असतो.
- Cloud platforms, जिथे user requests process करण्यासाठी वेगवेगळे servers वापरले जातात.

Distributed Algorithms चे महत्त्व:

1. **Scalability (वाढवण्याची क्षमता):**
   मोठ्या systems वर काम करण्यासाठी हे algorithms उपयोगी आहेत.

उदा: Social media platforms जसे की Facebook किंवा Twitter.

2. **Fault Tolerance (त्रुटी सहनक्षमता):**
   काही nodes failure झाले तरी system चा कार्यप्रवाह थांबत नाही.

   उदा: Google search engine मध्ये काही servers down झाले तरी search काम करत राहते.

3. **Resource Sharing (संसाधनांचे वाटप):**
   Resources (CPU, memory) efficiently वापरण्यासाठी distributed algorithms वापरले जातात.

Distributed Algorithms कसे काम करतात?

1. **Communication (संचार):**
   Nodes एकमेकांशी message passing द्वारे संपर्क साधतात.
   - "सर्व nodes एकत्र काम करण्यासाठी message sending वर अवलंबून असतात."
2. **Coordination (समन्वय):**
   Nodes आपापले काम कसे divide करतील आणि result कसे share करतील याचा समन्वय साधतात.
3. **Consensus (एकमत):**
   Distributed system मधील सर्व nodes एका निर्णयावर येण्यासाठी algorithm वापरतात.

   उदा: Blockchain मध्ये consensus mechanism.

Distributed Algorithms चे Types:

1. **Leader Election Algorithms:**
   System मधील एक leader node निवडण्यासाठी.
   - उदा: Ring-based algorithm.
2. **Mutual Exclusion Algorithms:**
   Resources वर exclusive access मिळवण्यासाठी.
   - "एका वेळी फक्त एकाच node ला resource वापरण्याची परवानगी देते."
3. **Consensus Algorithms:**
   Nodes एकत्र येऊन decision-making साठी काम करतात.
   - उदा: Paxos, Raft.
4. **Shortest Path Algorithms:**
   Network मधील दोन nodes दरम्यानचा shortest path शोधण्यासाठी.
   - उदा: Dijkstra algorithm.

Distributed Algorithms चे Advantage:

1. Large-scale systems handle करणे सोपे होते.
2. Fault-tolerant systems तयार करता येतात.
3. Data आणि computation effectively divide केले जाते.

Limitations:

1. **Latency (विलंब):** Messages पास होण्यामुळे process थोडा slow होतो.
2. **Complexity (जटिलता):** Algorithms design करणे कठीण असते.
3. **Failures:** Faults handle करण्यासाठी अधिक resources लागतात.

Distributed breadth first search,

Distributed BFS हा एक तंत्र आहे ज्यामध्ये **Breadth-First Search** (BFS) algorithm multiple systems किंवा nodes वर एकत्रितपणे कार्य करते. हा algorithm मोठ्या graphs process करण्यासाठी वापरला जातो, जिथे centralised processing practical नसते.

BFS चा उद्देश:

1. Graph मधील सर्व nodes (vertices) एका specific level नुसार explore करणे.
2. Source node पासून प्रत्येक node पर्यंतचा shortest path शोधणे.

Distributed BFS मध्ये, हाच process **multiple nodes** वर parallel चालतो.

Distributed BFS कसा काम करतो?

1. Graph Partitioning (ग्राफ विभाजन):

- Graph चे वेगवेगळ्या partitions मध्ये विभाजन केले जाते, आणि प्रत्येक partition एका node (system) वर assign केला जातो.
- Example: Graph मध्ये 1000 nodes असतील तर 10 systems 100-100 nodes process करतील.

2. Message Passing (संदेश प्रेषण):

- एक node दुसऱ्या node कडे संदेश (message) पाठवते. हे messages current level वरच्या nodes ची माहिती contain करतात.
- "Message passing मुळे nodes आपसात coordination करू शकतात."

3. Parallel Exploration (समानांतर अन्वेषण):

- सर्व nodes त्यांच्याकडे assign झालेले tasks independent पद्धतीने process करतात.
- प्रत्येक level complete झाल्यावर पुढील nodes चा exploration सुरू होतो.

4. Synchronization (सिंक साधणे):

- Nodes मधील communication sync करण्यासाठी barriers किंवा acknowledgements वापरले जातात.

Algorithm चे Steps:

1. **Initialization (सुरुवात):**
   - Source node initialized होते, आणि ती इतर nodes कडे message broadcast करते.
2. **Level-wise Exploration (level नुसार अन्वेषण):**
   - प्रत्येक node त्याच्या शेजारील nodes ला explore करते आणि माहिती पुढे पाठवते.
3. **Termination (समाप्ती):**
   - सर्व levels explore झाल्यावर algorithm पूर्ण होते.

Distributed BFS चे Advantage:

1. **Scalability (वाढवण्याची क्षमता):**
   - Large graphs effectively process होतात कारण workload multiple systems वर divide होते.
2. **Fault Tolerance (त्रुटी सहनशीलता):**
   - एका node मध्ये problem आली तरी इतर nodes process सुरू ठेवू शकतात.
3. **Speed (वेग):**
   - Parallel processing मुळे BFS जलद होते.

Distributed BFS च्या Limitatons:

1. **Communication Overhead (संदेशाचा ओझा):**
   - Systems मध्ये message passing जास्त असेल तर network traffic वाढतो.
2. **Synchronization Issues (सिंक समस्या):**
   - Nodes योग्य वेळेत synchronized नसतील तर process inefficient होते.
3. **Load Balancing (कामाचे संतुलन):**
   - काही nodes वर जास्त load येऊ शकतो, तर काही nodes idle राहू शकतात.

Example:

समजा एका social network graph मध्ये users आहेत आणि तुम्हाला एका specific user पर्यंतचा shortest path शोधायचा आहे.
Distributed BFS वापरून, users ची माहिती अनेक servers वर ठेवली जाऊ शकते. BFS प्रत्येक server वर parallel चालेल, आणि परिणाम लवकर मिळेल.

Distributed Minimum SpanningTree.

Distributed Minimum Spanning Tree (DMST) हा एक algorithmic approach आहे, जो distributed systems मध्ये spanning tree निर्माण करण्यासाठी वापरला जातो. Spanning Tree म्हणजे **connected graph** मधील minimum edge weight असलेला subgraph ज्यात cycle नसतो.

Distributed System मध्ये DMST कसा काम करतो?

Distributed system मध्ये प्रत्येक node ला information ची पूर्ण माहिती नसते; त्याऐवजी, त्यांच्याकडे फक्त शेजारी nodes ची माहिती असते. म्हणूनच **DMST** मध्ये nodes एकत्र काम करून spanning tree तयार करतात.

DMST Algorithm ची Characters:

1. **Decentralized Computation (वितरित गणना):**
   - प्रत्येक node आपले computations करते आणि फक्त शेजारी nodes सोबत communicate करते.
   - Example: प्रत्येक node आपल्याजवळची edge weight तपासते आणि decision घेते.
2. **Global Knowledge नसेल:**
   - पूर्ण graph बद्दल कोणत्याही एका node कडे माहिती नसते.
3. **Parallelism (सामांतरता):**
   - सर्व nodes एकाच वेळी काम करू शकतात, ज्यामुळे computation लवकर होते.

Algorithm चे Main Stages:

1. **Initiation (सुरुवात):**
   - प्रत्येक node स्वतःला स्वतंत्र component मानते.
   - "प्रत्येक node सुरुवातीला स्वतंत्र group मध्ये असतो."
2. **Edge Selection (काठांची निवड):**
   - प्रत्येक component त्याचा minimum edge शोधतो.
   - Example: एक node शेजारच्या nodes कडे पाहून सर्वात कमी weight असलेली edge निवडतो.
3. **Merge Components (गट विलीन करणे):**
   - Minimum edge वापरून दोन components एकत्र येतात.
   - "दोन गट minimum edge वापरून जोडले जातात."
4. **Repeat (पुनरावृत्ती):**
   - Process तोपर्यंत repeat होते जोपर्यंत सर्व components एकाच tree मध्ये विलीन होत नाहीत.

Examples:

समजा, **Graph G** मध्ये 6 nodes (A, B, C, D, E, F) आणि खालील weights असलेल्या edges आहेत:

Edge	Weight
A-B	4
B-C	6
A-D	3
D-E	2
E-F	5
C-F	4

- **Step 1:** Nodes स्वतःला components म्हणून consider करतात.
- **Step 2:** प्रत्येक node त्याच्या शेजारीकडे पाहून minimum edge निवडतो.
- **Step 3:** Components merge होतात, आणि process चालू राहतो.
- Result: A-D-E-F-C-B हा spanning tree तयार होतो, ज्याचा weight **20** आहे.

DMST चे Advantage:

1. **Scalability (वाढवण्याची क्षमता):** मोठ्या networks साठी उपयुक्त.
2. **Fault Tolerance (त्रुटि सहनशीलता):** काही nodes fail झाले तरी बाकी nodes काम करू शकतात.
3. **Parallelism:** वेळ कमी लागतो कारण अनेक nodes एकत्र काम करतात.

Limitations:

1. **Complex Communication (संवाद):** Nodes मध्ये जास्त communication लागते.
2. **Synchronization (सिंक करण्याचा प्रश्न):** Nodes synchronize करण्यासाठी अतिरिक्त overhead लागतो.

DMST चा Use:

- **Distributed Networks:** Internet, Telecommunication systems.
- **IoT (Internet of Things):** Sensor networks मधील data efficiently share करण्यासाठी.
- **Power Grids:** Minimum spanning trees वापरून power distribution optimize करणे.

String Matching-

Introduction,

String Matching म्हणजे एक algorithmic प्रक्रिया जी दोन strings मधील समानतेचे किंवा pattern ची शोध घेते. साधारणतः, एक मुख्य string आणि एक pattern string असतो. Pattern string ला मुख्य string मध्ये शोधून त्याचे occurences शोधले जातात. हे विविध applications मध्ये वापरले जाते, जसे की **text searching**, **data mining**, **bioinformatics**, आणि **information retrieval**.

String Matching चे मुख्य उद्दिष्ट:

- **Pattern चे स्थान शोधणे**: जर मुख्य string मध्ये pattern आहे, तर त्याचे स्थान शोधणे.
- **String comparison**: दोन strings एकमेकांशी तुलना करणे.

Algorithms for String Matching:

1. **Naive String Matching**:
    - सर्वात सोपा approach आहे. Pattern चा प्रत्येक character मुख्य string च्या प्रत्येक character सोबत compare केला जातो.

- Complexity: O(n×m) जिथे n म्हणजे मुख्य string ची length आणि m म्हणजे pattern ची length.
- "हा algorithm खूप साधा आहे पण मोठ्या strings साठी खूप वेळ घेतो."

2. **Knuth-Morris-Pratt (KMP) Algorithm**:
   - KMP algorithm pattern च्या पुनरावृत्त भागांचा उपयोग करून शोध प्रक्रियेला वेगवृद्धी करतो. यामध्ये **partial match table** तयार केली जाते.
   - Complexity: O(n+m)
   - "हे algorithm naive पेक्षा खूप जलद आहे आणि patterns ची पुन्हा तुलना टाळते."
3. **Rabin-Karp Algorithm**:
   - Rabin-Karp एक **hashing** बेस्ड algorithm आहे. प्रत्येक substring ची hash value pattern च्या hash value सोबत compare केली जाते.
   - Average case complexity: O(n+m), Worst case: O(n×m) "हे algorithm खूप जलद आहे, पण worst-case मध्ये naive पेक्षा स्लो होऊ शकतो."
4. **Boyer-Moore Algorithm**:
   - Pattern ची comparison मुख्य string च्या end पासून सुरु केली जाते, आणि pattern मधील character च्या **bad character rule** आणि **good suffix rule** वापरून searching प्रोसेस गती केली जाते.
   - Complexity: Best case O(n/m), Worst case O(n×m)
   - "हे algorithm pattern च्या structure नुसार शोध प्रक्रिया जास्त efficient बनवते."
5. **Aho-Corasick Algorithm**:
   - हे algorithm multiple patterns शोधण्यासाठी उपयुक्त आहे. अनेक patterns एकाच वेळी main string मध्ये शोधले जातात.
   - Complexity: O(n+k), जिथे k म्हणजे pattern च्या total length.
   - "Multiple patterns चे simultaneous matching साठी हे एक उत्तम algorithm आहे."

Applications of String Matching:

1. **Text Search**: Web browsers मध्ये keyword शोधणे.
2. **DNA Sequence Matching**: Bioinformatics मध्ये DNA sequences मध्ये pattern शोधणे.
3. **Spell Checking**: Spell checkers मध्ये शब्दांची तुलना करणे.
4. **Plagiarism Detection**: Documents मध्ये समान segments शोधणे.

The Naive string matching algorithm,

Naive string matching हा एक साधा आणि प्राथमिक algorithm आहे जो एका **pattern** (साधारणत: शोधायचा शब्द किंवा वाक्य) ला **text** (मुख्य स्ट्रिंग) मध्ये शोधतो. हा algorithm थोडा साधा आहे, पण त्याची कार्यक्षमता मोठ्या डेटावर कमी असू शकते.

कसे कार्य करते?

1. **Text आणि Pattern चे आकार**:
   Pattern चा आकार mmm आणि text चा आकार nnn असतो.

2. **साधा शोध प्रक्रिया**:
   प्रत्येक text च्या substring ला pattern शी compare केले जाते.
   - आपण text च्या प्रत्येक character पासून pattern च्या सर्व characters compare करतो.
   - जर सर्व characters जुळले, तर match सापडतो.
3. **Steps**:
   - Text च्या पहिल्या character पासून pattern ची तुलना सुरू करा.
   - प्रत्येक character वर, pattern च्या प्रत्येक character सोबत तुलनेत चला.
   - जर सर्व characters match झाले, तर pattern सापडले.
   - जर कुठे mismatch झाला, तर पुढील character पासून तुलना सुरू करा.

Algorithm:

1. Pattern आणि text मिळवा.
2. Text मध्ये सर्व starting positions च्या substrings चे pattern शी compare करा.
3. जर pattern पूर्णपणे match झाला तर त्या position वर pattern सापडले असे मानले जाते.

Example:
Text: **"ABC ABCDAB ABCDABCDABDE"**
Pattern: **"ABCDABD"**

- पहिल्या character पासून pattern शोधायला सुरुवात करा.
- **"ABC"** मध्ये तुलना होईल, आणि pattern पूर्णपणे match होईल.

Complexity:

- **Time Complexity**:
  Time complexity O((n−m+1)·m)O((n-m+1) \cdot m)O((n−m+1)·m), जिथे nnn text चे length आहे आणि mmm pattern चे length आहे.
  - प्रत्येक text position वर pattern चा पूर्ण तपास केला जातो, ज्यामुळे worst-case scenario मध्ये algorithm धीमा होतो.
- **Space Complexity**:
  Space complexity O(1)O(1)O(1), कारण अतिरिक्त memory ची आवश्यकता नाही. फक्त दोन pointers (text आणि pattern साठी) आवश्यक आहेत.

फायदे:

1. **साधे आणि अंतीमरीत**:
   Implementation खूप सोपे आणि थोड्या वेळात पूर्ण होऊ शकते.
2. **ठिकाणी उपयुक्त**:
   छोटे inputs आणि सिंपल प्रोग्राम्ससाठी योग्य आहे.

मर्यादा:

1. **कमी कार्यक्षमतेचे**:
   मोठ्या आकाराच्या text आणि pattern साठी, Time Complexity जास्त आहे. त्यामुळे याचे प्रदर्शन कमी होते.
2. **Optimizations ची आवश्यकता**:
   Naive algorithm पेक्षा अधिक कार्यक्षम algorithms (जसे की KMP, Rabin-Karp) आहेत, जे अधिक वेगवान कार्य करतात.

The Rabin-Karp algorithm

Rabin-Karp algorithm एक **string matching algorithm** आहे, जो **pattern matching** साठी वापरला जातो. याचा मुख्य उपयोग **large text** मध्ये दिलेल्या **pattern** चा शोध घेण्यासाठी केला जातो.

काम कसे करते?

Rabin-Karp algorithm मध्ये, प्रत्येक substring चा **hash value** तयार केला जातो आणि त्या hash value च्या मदतीने **pattern matching** केली जाते. याचे मुख्य advantages म्हणजे, **efficient matching** आणि **quick searching**.

तीन प्रमुख स्टेप्स:

1. **Hashing the pattern**:
   - Pattern च्या प्रत्येक character चा एक **hash value** तयार केला जातो.
   - Hash function वापरून pattern च्या hash value ला numeric रूपांतरण केले जाते.
2. **Hashing the substrings of the text**:
   - Text मधील प्रत्येक substring (ज्या length मध्ये pattern आहे) ची hash value काढली जाते.
   - Hash values compare केल्या जातात. जर hash values match केल्या, तर pattern आणि substring तुलना केली जाते.
3. **Pattern matching**:
   - जर substring च्या hash value pattern च्या hash value शी जुळली, तर substring आणि pattern एकमेकांशी जुळत असल्याचे तपासले जाते.
   - **Exact match** साठी, characters compare करणे आवश्यक असते, कारण hash collision होऊ शकतात.

Algorithm चे फायद

1. **Efficient Search**:
   - Rabin-Karp algorithm मध्ये pattern matching साठी **hashing** वापरले जाते, ज्यामुळे अनेक comparisons कमी होऊ शकतात.
   - Average case मध्ये, जेव्हा **hash collisions** कमी होतात, तेव्हा algorithm चा runtime **O(n)** होतो.
2. **Multiple Pattern Matching**:
   - Rabin-Karp algorithm ला एकाच वेळी **multiple patterns** शोधण्यासाठी extend केले जाऊ शकते, ज्यामुळे वेगवेगळ्या patterns साठी search प्रक्रिया वेगळी केली जाते.

Hash Function:

Rabin-Karp algorithm मध्ये **hash function** एक महत्त्वाचा भाग असतो. साधारणपणे, एका **rolling hash function** चा वापर केला जातो, ज्यात एक substring च्या hash value काढताना त्याचे values update केले जातात.

Hash Function चे Example: For a string of length nnn, its hash can be computed as:

$$\text{Hash}(text) = (text[0] \times d^{n-1} + text[1] \times d^{n-2} + \cdots + text[n-1] \times d^0) \mod q$$

जहां:

- d हा alphabet size (e.g., 256 for ASCII characters)
- q हा prime number (used to reduce hash value overflow)

Algorithm Complexity:

- **Best case (No collisions)**: O(n + m), where `n` is the length of the text and `m` is the length of the pattern.
- **Worst case (All hash values collide)**: O(n * m), which occurs when there are many hash collisions, and each potential match needs to be verified character by character.

मराठीत समजावून सांगितल्यास:

Rabin-Karp algorithm हे **string matching** चे एक प्रभावी साधन आहे, जे **text** मध्ये दिलेल्या **pattern** ची ओळख लावण्यासाठी **hashing technique** वापरते.

- त्यामध्ये pattern च्या आणि text च्या प्रत्येक substring च्या **hash values** तुलना केल्या जातात.
- जर hash values जुळल्या, तर त्याची characters तुलना केली जाते, कारण hash collision होऊ शकतो.