1. Алгоритм Фокса паралельного множення матриць.

Алгоритм Фокса для паралельного множення матриць є ефективним методом для обчислення добутку матриць. Алгоритм використовує метод блокування матриць та розподіл блоків між процесами для паралельного виконання. Розглянемо його детально крок за кроком:

- Дано дві матриці A і B, розміром N×N, які потрібно помножити, і результатом буде матриця C також розміром N×N.
- Розбиваємо матриці A і B на менші підматриці (блоки). Припустимо, що ми маємо P процесів, і P є квадратом деякого числа q (тобто P = q^2).
- Матриці ділимо на блоки розміром N/q×N/q. Кожен процесор отримує один блок кожної матриці.
- Створюються P процесів, кожен з яких відповідає за блоки A{i,j} і B{i,j}.
- Алгоритм проходить через q фаз (ітерацій). На кожній фазі виконується наступне:

1) Підготовка до множення:

На кожній фазі (k), процеси виконують комунікацію, щоб отримати потрібні блоки для множення. Наприклад, процес (i,j) отримує блок $A_{i,(j+i-k) \bmod q}$.

2) Множення блоків:

Кожен процес множить свій поточний блок матриці A на відповідний блок матриці B і додає результат до свого блоку матриці C:

$$C_{i,j}=C_{i,j}+A_{i,(j+i-k) \bmod q} \times B_{i,j}$$

3) Комунікація:

Після обчислення часткових результатів, блоки матриці B передаються сусіднім процесам.

- Після завершення всіх фаз, всі процеси завершують свою роботу, і результати з блоків матриці C об'єднуються в остаточну матрицю C.

2. Методи обміну повідомленнями стандарту MPI один до одного та їх застосування для розробки паралельних програм.

Методи обміну повідомленнями стандарту MPI (Message Passing Interface) є основою для розробки паралельних програм, що дозволяють обмінюватися даними між процесами у розподілених обчислювальних середовищах. Ось

детальний опис методів обміну повідомленнями "один до одного" (point-to-point communication) та їх застосування.

- MPI_Send: Використовується для відправки повідомлення з одного процесу до іншого.
- MPI_Recv: Використовується для прийому повідомлення від іншого процесу.
- MPI_Isend: Неблокуюча версія MPI_Send.
- MPI_Irecv: Неблокуюча версія MPI_Recv.
- MPI_Wait: Використовується для очікування завершення неблокуючої операції.
- MPI_Test: Використовується для перевірки завершення неблокуючої операції без блокування.

Методи MPI_Send і MPI_Recv використовуються для забезпечення прямого обміну даними між процесами. Це дозволяє розробникам явно контролювати передачу даних і синхронізацію між процесами.

Неблокуючі методи MPI_Isend і MPI_Irecv дозволяють процесам продовжувати обчислення, поки передача даних триває. Це підвищує ефективність, особливо у випадках, коли очікування завершення передачі може бути тривалим.

Методи "один до одного" також використовуються для розподілу роботи між процесами, де кожен процес виконує певну частину загального завдання і обмінюється проміжними результатами з іншими процесами. Це дозволяє збалансувати навантаження і зменшити час обчислень.

3. Напишіть реалізацію підрахунку середньої довжини слова за масивом слів, які зберігаються в файлі, з використанням Fork/Join фреймворку.

Приклад коду:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
```

```
public class Main {

    public static void main(String[] args) {
        String fileName = "example.txt";

        try {
            List<String> words = Files.readAllLines(Paths.get(fileName));
            ForkJoinPool pool = new ForkJoinPool();
            AverageWordLengthTask task = new AverageWordLengthTask(words, 0, words.size());
            Double averageLength = pool.invoke(task);

            System.out.println("Average word length: " + averageLength);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class AverageWordLengthTask extends RecursiveTask<Double> {
    private static final int THRESHOLD = 1000;
    private List<String> words;
    private int start;
    private int end;

    public AverageWordLengthTask(List<String> words, int start, int end) {
        this.words = words;
        this.start = start;
        this.end = end;
    }
```

```
    @Override
    protected Double compute() {
        int length = end - start;
        if (length <= THRESHOLD) {
            return computeDirectly();
        } else {
            int middle = start + length / 2;
            AverageWordLengthTask leftTask = new AverageWordLengthTask(words, start, middle);
            AverageWordLengthTask rightTask = new AverageWordLengthTask(words, middle, end);

            leftTask.fork();
            Double rightResult = rightTask.compute();
            Double leftResult = leftTask.join();

            return (leftResult * (middle - start) + rightResult * (end - middle)) / length;
        }
    }

    private Double computeDirectly() {
        int totalLength = 0;
        for (int i = start; i < end; i++) {
            totalLength += words.get(i).length();
        }
        return (double) totalLength / (end - start);
    }
}
```

Приклад вмісту “example.txt”:

```
pomegranate
persimmon
```

daddy

voyage

…..

4. Напишіть фрагмент коду OpenMPI/MPJ Express, який виконує паралельне обчислення елементів масиву C так, що кожний 1-ий елемент масиву C обчислюється як добуток середнього значення елементів i-ого рядка матриці A та середнього значення елементів i-ого рядка матриці B). В усіх масивах зберігаються дійсні числа. Розмір матриць A і B однаковий nxn.

Ось наведений код:

```
import java.util.Random
import mpi.*;

public class ParallelMatrixAverage {
   public static void main(String[] args) throws MPIException {
      MPI.Init(args);

      int rank = MPI.COMM_WORLD.Rank();
      int size = MPI.COMM_WORLD.Size();

      int n = 1000;

      double[][] A = new double[n][n];
      double[][] B = new double[n][n];
      double[] C = new double[n];



      if (rank == 0) {
Random random = new Random();
         for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
```

```
            A[i][j] = random.nextInt(100);
            B[i][j] = random.nextInt(100);
        }
    }
}

MPI.COMM_WORLD.Bcast(A, 0, A.length, MPI.OBJECT, 0);
MPI.COMM_WORLD.Bcast(B, 0, B.length, MPI.OBJECT, 0);

int rowsPerProcess = n / size;
int startRow = rank * rowsPerProcess;
int endRow = (rank + 1) * rowsPerProcess;

double[] partialC = new double[rowsPerProcess];

for (int i = startRow; i < endRow; i++) {
    double sumA = 0.0;
    double sumB = 0.0;
    for (int j = 0; j < n; j++) {
        sumA += A[i][j];
        sumB += B[i][j];
    }
    double avgA = sumA / n;
    double avgB = sumB / n;
    partialC[i - startRow] = avgA * avgB;
}

MPI.COMM_WORLD.Gather(partialC, 0, rowsPerProcess, MPI.DOUBLE, C,
0, rowsPerProcess, MPI.DOUBLE, 0);

if (rank == 0) {
    System.out.println("Resulting array C:");
    for (int i = 0; i < n; i++) {
```

```
            System.out.println("C[" + i + "] = " + C[i]);
        }
    }

    MPI.Finalize();
  }
}
```