# Documentație

## Analiza cerințelor

Obiectiv:  
Programul realizează adunarea numerelor mari reprezentate sub formă de vectori de cifre. Se poate opta între două metode de execuție:  
1. Secvențială  
2. Paralelă folosind MPI.

Intrare:  
- Două fișiere care conțin numerele mari de adunat, fiecare număr reprezentat ca un vector de cifre.  
- Parametrii de rulare specificați în linia de comandă.

Ieşire:  
- Fișier care conține rezultatul adunării.  
- Opțional: validarea rezultatului față de un fișier de așteptări (dacă se activează parametrul compare).

## Proiectare

### Structuri de Date

1. Vectori Unidimensionali (vector<int>)  
- Utilizare: Reprezintă numerele mari și rezultatul adunării.  
- Motiv: Permite manipularea eficientă a cifrelor numerelor mari și accesul rapid la poziții specifice.

2. String-uri (std::string)  
- Utilizare: Conversie între reprezentarea numerică și cea textuală a numerelor mari.  
- Motiv: Facilitează compararea și afișarea rezultatelor.

### Partiționare pe Procese

În modul paralel, datele sunt împărțite pe procese astfel:  
- Fiecare proces primește o parte din număr, calculează parțial adunarea și trimite rezultatul procesului rank = 0.  
- Procesul rank = 0 combină rezultatele parțiale și gestionează transporturile (carry) între segmente.

## Clase și Funcții

### Funcții Utilitare

1. vector<int> readNumber(const string& filename)  
- Descriere: Citește un număr mare dintr-un fișier și îl încarcă într-un vector de cifre.  
- Parametri: filename - calea fișierului de intrare.  
- Rezultate: Vector unidimensional care conține cifrele numărului.

2. string convertVectorToString(const vector<int>& vec)  
- Descriere: Transformă un vector de cifre într-un șir de caractere (string).  
- Parametri: vec - vectorul de cifre.  
- Rezultate: Reprezentarea numerică a vectorului sub formă de șir.

3. bool compareResults(const vector<int>& result, const string& expected)  
- Descriere: Compară rezultatul calculat cu rezultatul așteptat.  
- Parametri:  
 - result - vectorul de cifre rezultat.  
 - expected - rezultatul așteptat sub formă de string.  
- Rezultate: true dacă rezultatele coincid, false altfel.

4. void writeToFile(const vector<int>& number, const string& filename)  
- Descriere: Scrie un vector de cifre într-un fișier, fiecare cifră fiind separată prin spațiu.  
- Parametri:  
 - number - vectorul de cifre de scris.  
 - filename - numele fișierului de ieșire.

### Funcții Principale

1. vector<int> sequentialAddition(vector<int> num1, vector<int> num2)  
- Descriere: Realizează adunarea secvențială a două numere mari.  
- Parametri:  
 - num1 - primul număr mare.  
 - num2 - al doilea număr mare.  
- Rezultate: Vector care conține rezultatul adunării.

2. main(int argc, char\* argv[])  
- Descriere: Funcția principală care gestionează citirea inputului, alegerea metodei de execuție (secvențială/paralelă), rularea adunării și scrierea rezultatului.  
- Parametri:  
 - argc - numărul de argumente din linia de comandă.  
 - argv - vectorul de argumente.  
- Rezultate: Scrie rezultatul în fișier și afișează durata execuției.

### Funcții pentru Modul Paralel

1. Pe procesul rank = 0  
- Distributia datelor: Împarte cele două numere în segmente (chunk-uri) și le trimite proceselor worker.  
- Agregarea rezultatelor: Colectează rezultatele parțiale și gestionează transporturile (carry).

2. Pe procesele worker (rank > 0)  
- Primirea datelor: Primește segmentele corespunzătoare din cele două numere.  
- Calculul parțial: Realizează adunarea segmentului primit, gestionând transporturile interne.  
- Trimiterea rezultatului: Trimite rezultatul parțial procesului rank = 0.

## Specificații de Performanță

1. Complexitate Temporală:  
- Modul secvențial: O(n), unde n este numărul total de cifre.  
- Modul paralel: Aproximativ O(n/p) + O(p), unde p este numărul de procese MPI.

2. Complexitate Spațială:  
- Modul secvențial: O(n).  
- Modul paralel: Fiecare proces alocă doar segmentul său O(n/p).

### **Specificația Funcțiilor**

1. **vector<int> readNumber(const string& filename)**
   * **Parametri:**
     + filename - Calea fișierului care conține numărul.
   * **Rezultate:** Returnează un vector de cifre care reprezintă numărul mare citit.
2. **string convertVectorToString(const vector<int>& vec)**
   * **Parametri:**
     + vec - Vectorul de cifre ce trebuie convertit într-un șir.
   * **Rezultate:** Returnează un șir de caractere ce reprezintă numărul mare.
3. **bool compareResults(const vector<int>& result, const string& expected)**
   * **Parametri:**
     + result - Vector de cifre rezultat în urma calculului.
     + expected - Șir de caractere ce conține rezultatul așteptat.
   * **Rezultate:** Returnează true dacă rezultatele coincid, false altfel.
4. **vector<int> sequentialAddition(vector<int> num1, vector<int> num2)**
   * **Parametri:**
     + num1 - Primul număr mare reprezentat ca un vector de cifre.
     + num2 - Al doilea număr mare reprezentat ca un vector de cifre.
   * **Rezultate:** Returnează un vector de cifre care reprezintă rezultatul adunării.
5. **void writeToFile(const vector<int>& number, const string& filename)**
   * **Parametri:**
     + number - Vector de cifre ce conține numărul mare care trebuie scris.
     + filename - Calea fișierului în care va fi scris rezultatul.
   * **Rezultate:** Scrie vectorul de cifre în fișierul specificat, cifrele fiind separate prin spațiu.
6. **void writeToExpectedFile(const vector<int>& number, const string& filename)**
   * **Parametri:**
     + number - Vector de cifre ce conține rezultatul așteptat.
     + filename - Calea fișierului în care se va scrie rezultatul.
   * **Rezultate:** Scrie vectorul de cifre în format compact, fără spații, în fișierul specificat.
7. **string readExpected(const string& filename)**
   * **Parametri:**
     + filename - Calea fișierului care conține rezultatul așteptat.
   * **Rezultate:** Returnează șirul de caractere ce reprezintă rezultatul așteptat.
8. **void printNumber(const vector<int>& number)**
   * **Parametri:**
     + number - Vectorul de cifre ce reprezintă numărul mare.
   * **Rezultate:** Afișează numărul mare în consolă, eliminând zerourile de la începutul numărului.
9. **int main(int argc, char\* argv[])**
   * **Parametri:**
     + argc - Numărul de argumente din linia de comandă.
     + argv - Vector de șiruri de caractere ce conțin argumentele din linia de comandă.
   * **Rezultate:**
     + Inițializează MPI, alege modul de execuție (secvențial sau paralel), realizează calculul și scrie rezultatul în fișier.
     + Afișează durata execuției și validează rezultatul dacă parametrul compare este activat.

### **3. Detalii de Implementare**

Implementarea soluției pentru programul de adunare a numerelor mari implică mai mulți pași esențiali, de la citirea datelor din fișiere până la executarea operațiilor de adunare (secvențială sau paralelă) și scrierea rezultatelor într-un fișier de ieșire.

#### **1. Citirea Datelor din Fișiere**

Funcția **readNumber()** este responsabilă pentru citirea unui număr mare dintr-un fișier. Aceasta verifică dacă fișierul poate fi deschis și încarcă datele într-un vector de cifre.

vector<int> readNumber(const string& filename) {  
 ifstream fin(filename);  
 int size;  
 fin >> size;  
 vector<int> number(size);  
 for (int i = 0; i < size; i++) {  
 fin >> number[i]; // Read digits   
 }  
 return number;  
}

* **Input:** Un fișier text cu formatul:

4

1 2 3 4

Reprezintă numărul **1234**.

* **Output:** Un vector de cifre: [1, 2, 3, 4].

#### **2. Adunarea Secvențială**

Funcția **sequentialAddition()** realizează adunarea numerelor mari, gestionând transporturile între cifre.

vector<int> sequentialAddition(vector<int> num1, vector<int> num2) {  
 int maxSize = max(num1.size(), num2.size());  
 vector<int> result(maxSize + 1, 0);  
 int carry = 0;  
  
 for (int i = 0; i < maxSize; ++i) {  
 int digit1 = (i < num1.size()) ? num1[i] : 0;  
 int digit2 = (i < num2.size()) ? num2[i] : 0;  
 int sum = digit1 + digit2 + carry;  
 result[i] = sum % 10;  
 carry = sum / 10;  
 }  
  
 if (carry) {  
 result[maxSize] = carry;  
 } else {  
 result.pop\_back();  
 }  
  
 return result;  
}

* **Input:** Doi vectori: [4, 3, 2, 1] și [6, 5, 4] (reprezentând numerele 1234 și 456).
* **Output:** Vectorul [0, 9, 7, 1] (reprezentând numărul 1690).

#### **3. Adunarea Paralelă cu Send/Recv**

În modul parallel, cu send/recieve, datele sunt împărțite pe segmente între procesele MPI.

* Procesul **rank = 0** împarte numerele în segmente și trimite fiecărui proces worker.
* Procesele worker efectuează adunarea segmentelor lor.
* Rezultatele parțiale sunt returnate procesului **rank = 0**, care gestionează transporturile globale.

##### **Distribuirea Datelor:**

Cod pe procesul **rank = 0** pentru a trimite segmentele către procesele worker:

if (rank == 0) {  
 number1 = readNumber(filename1);  
 number2 = readNumber(filename2);  
  
 int maxSize = max(number1.size(), number2.size());  
 int chunkSize = (maxSize + size - 2) / (size - 1); // Distribute work among workers  
  
 auto start = chrono::high\_resolution\_clock::*now*();  
  
 for (int i = 0; i < maxSize; i += chunkSize) {  
 int actualChunkSize = min(chunkSize, maxSize - i);  
  
 vector<int> chunk1(actualChunkSize, 0), chunk2(actualChunkSize, 0);  
  
 for (int j = 0; j < actualChunkSize; ++j) {  
 if (i + j < number1.size()) chunk1[j] = number1[i + j];  
 if (i + j < number2.size()) chunk2[j] = number2[i + j];  
 }  
  
 MPI\_Send(&actualChunkSize, 1, MPI\_INT, (i / chunkSize) + 1, 0, MPI\_COMM\_WORLD);  
 MPI\_Send(chunk1.data(), actualChunkSize, MPI\_INT, (i / chunkSize) + 1, 1, MPI\_COMM\_WORLD);  
 MPI\_Send(chunk2.data(), actualChunkSize, MPI\_INT, (i / chunkSize) + 1, 2, MPI\_COMM\_WORLD);  
 }  
  
 result.resize(maxSize + 1, 0);  
 for (int i = 1; i < size; ++i) {  
 int actualChunkSize;  
 MPI\_Recv(&actualChunkSize, 1, MPI\_INT, i, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
  
 vector<int> partialResult(actualChunkSize + 1, 0);  
 MPI\_Recv(partialResult.data(), actualChunkSize + 1, MPI\_INT, i, 4, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
  
 for (int j = 0; j < actualChunkSize; ++j) {  
 result[(i - 1) \* chunkSize + j] += partialResult[j];  
 }  
 }  
  
 int carry = 0;  
 for (int i = 0; i < result.size(); ++i) {  
 result[i] += carry;  
 carry = result[i] / 10;  
 result[i] %= 10;  
 }  
 if (carry) {  
 result.push\_back(carry);  
 }  
  
 writeToFile(result, "Numar3.txt");  
 auto stop = chrono::high\_resolution\_clock::*now*();  
 chrono::duration<double, std::milli> duration = stop - start;  
 cout << duration.count() << " milliseconds" << endl;  
}

**Calculul pe Procese Worker:**

Cod pentru procesele worker care primesc segmente, efectuează adunarea și trimit rezultatul înapoi:

int actualChunkSize;  
MPI\_Recv(&actualChunkSize, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
  
vector<int> chunk1(actualChunkSize, 0), chunk2(actualChunkSize, 0), partialResult(actualChunkSize + 1, 0);  
  
MPI\_Recv(chunk1.data(), actualChunkSize, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
MPI\_Recv(chunk2.data(), actualChunkSize, MPI\_INT, 0, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  
  
int carry = 0;  
for (int i = 0; i < actualChunkSize; ++i) {  
 int sum = chunk1[i] + chunk2[i] + carry;  
 partialResult[i] = sum % 10;  
 carry = sum / 10;  
}  
partialResult[actualChunkSize] = carry;  
  
MPI\_Send(&actualChunkSize, 1, MPI\_INT, 0, 3, MPI\_COMM\_WORLD);  
MPI\_Send(partialResult.data(), actualChunkSize + 1, MPI\_INT, 0, 4, MPI\_COMM\_WORLD);

***4. Adunarea Paralelă*** *cu Scatter/Gather*

Codul utilizează funcțiile **MPI\_Scatter** și **MPI\_Gather** pentru a distribui segmentele și a colecta rezultatele în cadrul unui proces de adunare paralelă. Această abordare simplifică gestionarea datelor, utilizând metode predefinite din MPI pentru trimiterea și colectarea datelor între procese.

1. **Distribuirea Datelor:**

* **MPI\_Scatter**:
  + Imparte un vector de date (chunk1, chunk2) în segmente și le distribuie între toate procesele.
* **MPI\_Gather**:
  + Colectează rezultatele parțiale de la toate procesele într-un singur vector (gatheredResult), în procesul root (rank 0).

auto start = chrono::high\_resolution\_clock::*now*();  
number1 = readNumber(filename1);  
number2 = readNumber(filename2);  
  
int maxSize = max(number1.size(), number2.size());  
int chunkSize = (maxSize + size - 1) / size; // Distribute work among all processes  
  
vector<int> chunk1(size \* chunkSize, 0), chunk2(size \* chunkSize, 0), partialResult(chunkSize + 1, 0);  
vector<int> recvChunk1(chunkSize, 0), recvChunk2(chunkSize, 0);  
  
if (rank == 0) {  
 for (int i = 0; i < number1.size(); ++i) {  
 chunk1[i] = number1[i];  
 }  
 for (int i = 0; i < number2.size(); ++i) {  
 chunk2[i] = number2[i];  
 }  
}  
  
MPI\_Scatter(chunk1.data(), chunkSize, MPI\_INT, recvChunk1.data(), chunkSize, MPI\_INT, 0, MPI\_COMM\_WORLD);  
MPI\_Scatter(chunk2.data(), chunkSize, MPI\_INT, recvChunk2.data(), chunkSize, MPI\_INT, 0, MPI\_COMM\_WORLD);  
  
int carry = 0;  
for (int i = 0; i < chunkSize; ++i) {  
 int sum = recvChunk1[i] + recvChunk2[i] + carry;  
 partialResult[i] = sum % 10;  
 carry = sum / 10;  
}  
partialResult[chunkSize] = carry;  
  
vector<int> gatheredResult((chunkSize + 1) \* size, 0);  
MPI\_Gather(partialResult.data(), chunkSize + 1, MPI\_INT, gatheredResult.data(), chunkSize + 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

result.resize(maxSize + 1, 0);

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

for (int j = 0; j < chunkSize; ++j) {

result[i \* chunkSize + j] += gatheredResult[i \* (chunkSize + 1) +j];

}

}

int carry = 0;

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

result[i] += carry;

carry = result[i] / 10;

result[i] %= 10;

}

if (carry) {

result.push\_back(carry);

}

writeToFile(result, "Numar3.txt");

auto stop = chrono::high\_resolution\_clock::now();

chrono::duration<double, std::milli> duration = stop - start;

cout << duration.count() << " milliseconds" << endl;

### **Fluxul General al Algoritmului**

#### a) **Procesul Rank = 0 (Root)**

* Inițializează vectorii de intrare (chunk1, chunk2) cu datele numerelor mari.
* Apelează **MPI\_Scatter** pentru a distribui segmentele către toate procesele.
* După ce primește rezultatele de la procese utilizând **MPI\_Gather**, combină rezultatele parțiale, gestionează transportul global (carry) și construiește rezultatul final.
* Scrie rezultatul în fișier și calculează timpul de execuție.

#### b) **Toate Procesele (Inclusiv Rank = 0)**

* Primesc segmentele de date folosind **MPI\_Scatter**.
* Efectuează adunarea locală pentru segmentele lor.
* Transmit rezultatele parțiale (incluzând transportul) către procesul root folosind **MPI\_Gather**.

### 3. **Fluxul Detaliat**

#### **a) Distribuirea Datelor (Scatter)**

* Procesul root inițializează vectorii chunk1 și chunk2 cu datele numerelor mari, completând valorile lipsă cu zero pentru a potrivi dimensiunea vectorilor.
* Se calculează dimensiunea fiecărui segment (chunkSize), astfel încât datele să fie distribuite uniform între procese.
* **MPI\_Scatter** distribuie segmentele corespunzătoare fiecărui proces în vectori locali (recvChunk1, recvChunk2).

#### **b) Calculul Local**

* Fiecare proces efectuează adunarea locală pe segmentele primite:
  + Adună cifrele corespunzătoare din recvChunk1 și recvChunk2.
  + Gestionează transportul (carry) între cifre.
  + Rezultatul local este stocat în partialResult, care include transportul final.

#### **c) Colectarea Rezultatelor (Gather)**

* **MPI\_Gather** colectează rezultatele parțiale de la toate procesele într-un singur vector (gatheredResult) în procesul root.

#### **d) Asamblarea Rezultatelor**

* Procesul root reconstruiește rezultatul final:
  + Adaugă rezultatele parțiale în vectorul final result, gestionând transportul global între segmente.
  + Dacă există un transport rămas, acesta este adăugat ca o cifră suplimentară.
* Rezultatul este scris într-un fișier.

## Cazuri de Testare

### Intrări de Test

1. Fișierele Numar1T1.txt și Numar2T1.txt conținând numere cu 18 cifre.
2. Fișierele Numar1T2.txt și Numar2T2.txt conținând numere cu 1000 cifre.
3. Fișierele Numar1T3.txt și Numar2T3.txt conținând numere cu 100, respectiv 100000 cifre.
4. Fișierele Numar1T4.txt și Numar2T4.txt conținând numere cu 16 cifre.

### Scenarii

1. Execuție Secvențială:  
- Dimensiuni: 18-18, 1000-1000, 100-100000 și 16-16  
- Output: Timp de execuție exprimat în milisecunde și fișier rezultat Numar3.txt.

2. Execuție Paralelă:  
- Număr variabil de procese: 4, 8, 16.  
- Output: Timp de execuție exprimat în milisecunde și fișier rezultat.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Filename1 | Filename2 | Function name | Nr. Processes | Timp executie |
| Numar1T1.txt | Numar2T1.txt | secvential | 1 | 0.638 |
| Numar1T2.txt | Numar2T2.txt | secvential | 1 | 1.0328 |
| Numar1T3.txt | Numar2T3.txt | secvential | 1 | 17.5456 |
| Numar1T4.txt | Numar2T4.txt | secvential | 1 | 0.6408 |
| Numar1T1.txt | Numar2T1.txt | paralel | 4 | 0.8211 |
| Numar1T2.txt | Numar2T2.txt | paralel | 4 | 0.9683 |
| Numar1T2.txt | Numar2T2.txt | paralel | 8 | 1.2664 |
| Numar1T2.txt | Numar2T2.txt | paralel | 16 | 2.1591 |
| Numar1T3.txt | Numar2T3.txt | paralel | 4 | 12.2702 |
| Numar1T3.txt | Numar2T3.txt | paralel | 8 | 22.5031 |
| Numar1T3.txt | Numar2T3.txt | paralel | 16 | 14.7534 |
| Numar1T4.txt | Numar2T4.txt | paralel | 4 | 0.8114 |