При множенні матриці A на матрицю B отримаємо матрицю C, елементи якої обчислюються за формулою (усі матриці є квадратними порядку n):

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

Безпосередня реалізація на мові програмування C має вигляд:

```
for (int i = 0; i < n; i++)
	for (int j = 0; j < n; j++)
	{
		c[i][j] = 0;
		for (int k = 0; k < n; k++)
			c[i][j] = c[i][j] + a[i][k] * b[k][j];
	}
```

В цій програмі `c[i][j]` переписується багато разів з метою економії пам'яті. Таким чином, значення `c[i][j]` присвоюється більше одного разу. При перетворенні цієї ж програми в програму з одноразовим присвоюванням кількість індексів матриці C – зросте:

```
for (int i = 0; i < n; i++)
	for (int j = 0; j < n; j++)
	{
		c[i][j][0] = 0;
		for (int k = 0; k < n; k++)
			c[i][j][k+1] = c[i][j][k] + a[i][k] * b[k][j];
	}
```

Тепер, кожному елементу матриці C буде присвоєно лише одне значення, а остаточні значення будуть отримані на останньому кроці ітерації (результат – `c[i][j][n]`).

Рекурсивний алгоритм –це алгоритм, який визначається за допомогою правила одноразового присвоювання і є стислим представленням багатьох алгоритмів. Побудова рекурсивного алгоритму зводиться до виведення рекурсивних рівнянь. Дії паралельних алгоритмів адекватно описуються в рекурсивних рівняннях з просторово-часовими індексами якщо один індекс використовується для часу, а інші – для простору (надалі – індексний простір).

Для випадку множення матриці на матрицю, рекурсивне рівняння буде мати вигляд:

$c_{ij}^{(k+1)} = c_{ij}^{(k)} + a_{ik}^{(j)} * b_{kj}^{(i)}$, де

$a_{ik}^{(j)} = a[i][k][j]$ і $b_{kj}^{(i)} = b[k][j][i]$.

Граф залежностей (ГЗ) - це граф, який описує залежність обчислень в алгоритмі. ГЗ може розглядатися як графічне представлення алгоритму з одноразовим присвоєнням. ГЗ називається повним, якщо він визначає всі залежності між всіма змінними в індексному просторі.

Локалізований граф залежностей – алгоритм є локалізований, якщо всі змінні безпосередньо залежать лише від змінних в сусідніх вузлах. Дані, що пересилаються незмінними до всіх вершин графу називаються передаваними, в іншому випадку – це непередавані дані.

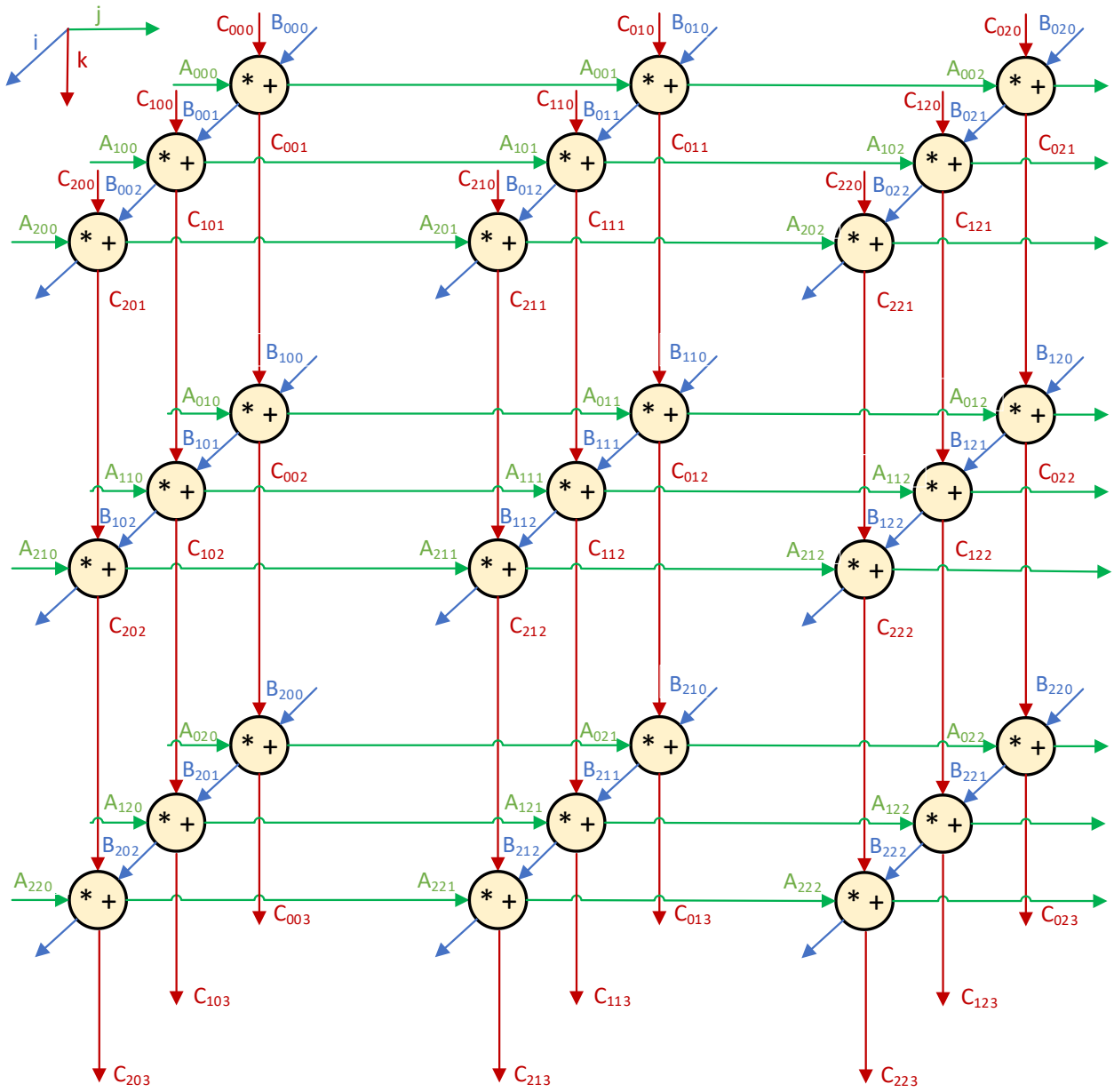

Рис. 1. Граф залежностей з локальними зв’язками для множення матриці на матрицю (при n = 3)

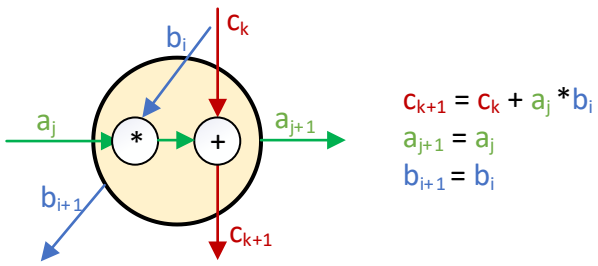

Рис. 2. Функціональна схема вузла графу

Програма для локалізованого алгоритму:

```
for (int i = 0; i < n; i++)
	for (int j = 0; j < n; j++)
	{
		c3D[i][j][0] = 0;
		a3D[i][j][0] = a[i][j];
		b3D[i][j][0] = b[i][j];
	}

for (int i = 0; i < n; i++)
	for (int j = 0; j < n; j++)
	{
		for (int k = 0; k < n; k++)
		{
			c3D[i][j][k + 1] = c3D[i][j][k] + a3D[i][k][j] * b3D[k][j][i];
			a3D[i][k][j + 1] = a3D[i][k][j];
			b3D[k][j][i + 1] = b3D[k][j][i];
		}
	}
```

Програма для рекурсивного локалізованого алгоритму:

```
void MultMatrixRecursAalgorithm(int*** c3D, int*** a3D, int*** b3D, int n)
{
	static int i = 0, j = 0, k = 0;

	if (i < n)
	{
		if (j < n)
		{
			if (k < n)
			{

				c3D[i][j][k + 1] = c3D[i][j][k] + a3D[i][k][j] *
b3D[k][j][i];
				a3D[i][k][j + 1] = a3D[i][k][j];
				b3D[k][j][i + 1] = b3D[k][j][i];

				k++;
				MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);
			}
			k = 0;
			j++;
			MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);
		}
		j = 0;
		i++;
		MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);
	}
}
```

```
//тестова програма
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#include <windows.h>

int** CreateMatrix(int n)
{
	int** a = new int* [n];
	for (int i = 0; i < n; i++)
		a[i] = new int[n];
	return a;
}

int*** CreateMatrix3D(int n)
{
	int*** a = new int** [n];
	for (int i = 0; i < n; i++)
	{
		a[i] = new int* [n];
		for (int j = 0; j < n; j++)
		{
			a[i][j] = new int[n + 1];
		}
	}
	return a;
}

void DeleteMatrix(int** a, int n)
{
	for (int i = 0; i < n; i++)
		delete[] a[i];
	delete[] a;
}

void DeleteMatrix3D(int*** a, int n)
{
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < n; j++)
			delete[] a[i][j];
		delete[] a[i];
	}
	delete[] a;
}

void FillMatrix(int** a, int n)
{
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
		{
			a[i][j] = rand() % 10;
		}

}

void InputMatrix(int** a, int n)
{
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < n; j++)
		{
			printf("Enter [%d][%d] element = ", i, j);
			scanf_s("%d", &(a[i][j]));
		}
```

```
			printf("\n");
		}
}

void PrintMatrix(int** a, int n)
{
		int i, j;
		for (i = 0; i < n; i++)
		{
			for (j = 0; j < n; j++)
				printf("%d\t", a[i][j]);
			printf("\n");
		}
		printf("\n");
}

int** Martix3Dto2D(int** a, int*** b, int n)
{
		int i, j;
		for (i = 0; i < n; i++)
		{
			for (j = 0; j < n; j++)
				a[i][j] = b[i][j][n];
		}
		return a;
}

void PrintMatrix3D(int*** a, int n)
{
		int i, j;
		for (i = 0; i < n; i++)
		{
			for (j = 0; j < n; j++)
				printf("%d\t", a[i][j][n]);
			printf("\n");
		}
		printf("\n");
}

void PrintMatrixToFile(FILE* f, char* title, int** a, int n)
{
		fprintf(f, title);
		fprintf(f, "\n");
		for (int i = 0; i < n; i++)
		{
			for (int j = 0; j < n; j++)
				fprintf(f, "%d\t", a[i][j]);
			fprintf(f, "\n");
		}
		fprintf(f, "\n");
}

// c = a * b
int** MultMatrix(int** c, int** a, int** b, int n)
{
		for (int i = 0; i < n; i++)
			for (int j = 0; j < n; j++)
			{
				c[i][j] = 0;
				for (int k = 0; k < n; k++)
					c[i][j] = c[i][j] + a[i][k] * b[k][j];
			}
		return c;
}

int** MultMatrixOneTime(int** c, int** a, int** b, int n)
{
		int*** c3D = CreateMatrix3D(n);
```

```
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
		{
			c3D[i][j][0] = 0;
			for (int k = 0; k < n; k++)
				c3D[i][j][k + 1] = c3D[i][j][k] + a[i][k] * b[k][j];
		}

	Martix3Dto2D(c, c3D, n);
	DeleteMatrix3D(c3D, n);

	return c;
}

int** MultMatrixLocalAlgorithm(int** c, int** a, int** b, int n)
{
	int*** c3D = CreateMatrix3D(n);
	int*** a3D = CreateMatrix3D(n);
	int*** b3D = CreateMatrix3D(n);

	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
		{
			c3D[i][j][0] = 0;
			a3D[i][j][0] = a[i][j];
			b3D[i][j][0] = b[i][j];
		}

	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
		{
			for (int k = 0; k < n; k++)
			{
				c3D[i][j][k + 1] = c3D[i][j][k] + a3D[i][k][j] *
b3D[k][j][i];
				a3D[i][k][j + 1] = a3D[i][k][j];
				b3D[k][j][i + 1] = b3D[k][j][i];
			}
		}

	Martix3Dto2D(c, c3D, n);

	DeleteMatrix3D(c3D, n);
	DeleteMatrix3D(a3D, n);
	DeleteMatrix3D(b3D, n);


	return c;
}

void MultMatrixRecursAalgorithm(int*** c3D, int*** a3D, int*** b3D, int n)
{
	static int i = 0, j = 0, k = 0;

	if (i < n)
	{
		if (j < n)
		{
			if (k < n)
			{

				c3D[i][j][k + 1] = c3D[i][j][k] + a3D[i][k][j] *
b3D[k][j][i];
				a3D[i][k][j + 1] = a3D[i][k][j];
				b3D[k][j][i + 1] = b3D[k][j][i];

				k++;
```

```
                        MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);
                }
                k = 0;
                j++;
                MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);
            }
            j = 0;
            i++;
            MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);
        }
}

int** MultMatrixRA(int** c, int** a, int** b, int n)
{
        int*** c3D = CreateMatrix3D(n);
        int*** a3D = CreateMatrix3D(n);
        int*** b3D = CreateMatrix3D(n);

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
            {
                c3D[i][j][0] = 0;
                a3D[i][j][0] = a[i][j];
                b3D[i][j][0] = b[i][j];
            }

        MultMatrixRecursAalgorithm(c3D, a3D, b3D, n);

        Martix3Dto2D(c, c3D, n);

        DeleteMatrix3D(c3D, n);
        DeleteMatrix3D(a3D, n);
        DeleteMatrix3D(b3D, n);

        return c;

}


int main(void)
{
        srand((unsigned int)time(NULL));

        int n;
        printf("Enter the matrix size n : ");
        scanf_s("%d", &n);

        int** A = CreateMatrix(n);
        int** B = CreateMatrix(n);


        int in;
        printf("\nSelect an input option : 1 - manual input, else  - random input : ");
        scanf_s("%d", &in);
        printf("\n");

        const char* fileName = "Results.txt";
        FILE* file = fopen(fileName, "w");
        if (file == NULL)
        {
            printf("\nError opening file. Check the path and permissions !\n");
            return 1;
        }

        if (in == 1)
        {
            // manual input matrix A
            printf("Enter the elements of matrix A :\n");
```

```
            InputMatrix(A, n);
            PrintMatrixToFile(file, (char*)"Matrix A = ", A, n);

            // manual input matrix B
            printf("Enter the elements of matrix B :\n");
            InputMatrix(B, n);
            PrintMatrixToFile(file, (char*)"Matrix B = ", B, n);

        }
        else
        {
            printf("Random input... \n\n"); \

            // random input matrix A
            FillMatrix(A, n);
            PrintMatrixToFile(file, (char*)"Matrix B = ", A, n);
            printf("Matrix A :\n");
            PrintMatrix(A, n);

            // random input matrix B
            FillMatrix(B, n);
            PrintMatrixToFile(file, (char*)"Matrix B = ", B, n);
            printf("Matrix B :\n");
            PrintMatrix(B, n);
        }

        int** C = CreateMatrix(n);

        MultMatrix(C, A, B, n);

        printf("Matrix C = A * B :\n");
        PrintMatrixToFile(file, (char*)"Matrix C = A * B = ", C, n);
        PrintMatrix(C, n);


        MultMatrixOneTime(C, A, B, n);

        printf("Matrix C = A * B, program with one-time assignment :\n");
        PrintMatrixToFile(file, (char*)"Matrix C = A * B = ", C, n);
        PrintMatrix(C, n);

        MultMatrixLocalAlgorithm(C, A, B, n);

        printf("Matrix C = A * B, program with the localized algorithm:\n");
        PrintMatrixToFile(file, (char*)"Matrix C = A * B = ", C, n);
        PrintMatrix(C, n);

        MultMatrixRA(C, A, B, n);

        printf("Matrix C = A * B, program with the recursive localized algorithm :\n");
        PrintMatrixToFile(file, (char*)"Matrix C = A * B = ", C, n);
        PrintMatrix(C, n);

        DeleteMatrix(A, n);
        DeleteMatrix(B, n);
        DeleteMatrix(C, n);

        fclose(file);

        return 0;
}
```