

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з дисципліни
«Паралельні та розподілені обчислення»
для студентів спеціальності 123 «Комп'ютерна інженерія»
всіх форм навчання
Частина 2

2020

Методичні вказівки до лабораторних робіт з дисципліни «Паралельні та розподілені обчислення» для студентів спеціальності 123 «Комп'ютерна інженерія» всіх форм навчання. Частина 2 / Укл.: Р.К. Кудерметов, В.В. Шкарупило, О.В. Польська. – Запоріжжя: НУ «Запорізька політехніка», 2020. – 54 с.

Укладачі:

Р.К. Кудерметов, доцент, к.т.н.
В.В. Шкарупило, доцент, к.т.н.
О.В. Польська, ст. викладач

Рецензент:

М.Ю. Тягунова, доцент, к.т.н.

Відповідальний
за випуск:

Р.К. Кудерметов, доцент, к.т.н.

Затверджено:
на засіданні кафедри

«Комп'ютерні системи та мережі»
Протокол № 5 від 29.01.2020 р.

Рекомендовано до видання:

НМК факультету КНТ
Протокол № 6 від 31.01.2020 р.

ЗМІСТ

Лабораторна робота №3 Паралельні обчислення з використанням технології MPI	4
3.1 Базові функції MPI	4
3.2 Функції колективного обміну MPI.....	7
3.3 Застосування MPI у практичних обчисленнях.....	10
3.3.1 Розподіл обчислень між процесами.....	10
3.3.2 Паралельні обчислення за допомогою методу Монте-Карло.....	18
3.4 Завдання до лабораторної роботи	24
3.5 Зміст звіту.....	27
3.6 Контрольні питання.....	27
Лабораторна робота №4 Паралельні обчислення з використанням технології OpenMP.....	28
4.1 Основні положення і поняття технології OpenMP	28
4.2 Застосування OpenMP для практичних обчислень.....	34
4.2.1 Бульбашкове послідовне сортування.....	34
4.2.2 Сортування методом «парний-непарний».....	38
4.2.3 Швидке сортування	43
4.3 Завдання до лабораторної роботи	47
4.4 Зміст звіту.....	51
4.5 Контрольні питання.....	51
Рекомендована література	52
Додаток А. Порядок створення MPI-додатку у середовищі DEV-C++	53
Додаток Б. Порядок запуску MPI-додатку за допомогою WMPIEXEC	54

ЛАБОРАТОРНА РОБОТА №3

ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MPI

Мета роботи – вивчити елементарні функції технології MPI і структуру простих паралельних програм на основі цієї технології.

3.1 Базові функції MPI

Частина функцій бібліотеки MPI призначена для створення структури MPI-програми і використовується практично у всіх програмах на основі технології MPI [3]. Перелічимо основні з них:

– `MPI_Init(int* argc, char*** argv)` – ініціалізує MPI-середовище, тобто створює початкову групу процесів і комунікатор `MPI_COMM_WORLD` – область зв'язку, що об'єднує усі процеси програмного додатку. Аргументами функції є параметри головної функції – точки входу в програму – `main(int argc, char* argv[])`;

– `MPI_Finalize()` – завершує усі процеси додатку і знищує усі комунікатори;

– для аварійного завершення роботи програми (наприклад, у випадку відсутності динамічної пам'яті для створення масиву) використовується функція `MPI_Abort(MPI_Comm comm, int err)`;

– `MPI_Comm_size(MPI_Comm comm, int *size)` – функція визначення кількості процесів у області зв'язку `comm`;

– `MPI_Comm_rank(MPI_Comm comm, int *rank)` – функція визначення номеру процесу в області зв'язку `comm`.

Для організації блокуючого двохточкового обміну даними між процесами використовуються наступні функції:

– передачі –

`MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`;

– прийому –

`MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`;

У наведених функціях фігурують наступні аргументи:

– `buf` – адреса буферу даних, що передаються (для функції передачі), чи буферу даних, що приймаються (для функції прийому);

– count – число даних типу datatype, що приймаються/передаються;
 – dest – номер процесу-одержувача повідомлення;
 – source – номер процесу-відправника повідомлення;
 – tag – ідентифікатор повідомлення;
 – comm – назва комунікатора;
 – status – структура з трьома полями (MPI_SOURCE, MPI_TAG і MPI_ERROR).

Про деякі інші базові функції бібліотеки MPI буде сказано у ході опису завдань до лабораторної роботи.

У наступному прикладі демонструється використання базових функцій MPI для створення структури паралельного додатку (лістинг 3.1).

Лістинг 3.1 – Використання базових функцій MPI

```

1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char *argv[])
5  {
6      int size, rank, i;
7      MPI_Init(&argc, &argv);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10     if (rank == 0)
11     {
12         printf("Number of processes = %d\n", size);
13     }
14     printf("My number in MPI_COMM_WORLD = %d\n", rank);
15     MPI_Finalize();
16     return 0;
17 }
```

У лістингу 3.2 наведено програму, в якій два процеси обмінюються даними.

Лістинг 3.2 – Програма з функціями двохточкового обміну даними

```

1  #include <stdio.h>
2  #include "mpi.h"
3  int main(int argc, char *argv[])
4  {
5      int size, rank, count;
6      double data[20];
7      MPI_Status status;
8      MPI_Init(&argc,&argv);
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11     if (size!=2)
12     {
13         if (rank ==0)
14             printf("Only 2 tasks required instead of %d,
15                     abort \n",size);
16         MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
17         return -1;
18     }
19     if (rank==0) MPI_Send(data,5,MPI_DOUBLE,1,100,
20         MPI_COMM_WORLD);
21     else
22     {
23         MPI_Recv(data,5,MPI_DOUBLE,0,100,MPI_COMM_WORLD,
24             &status);
25         MPI_Get_count(&status,MPI_DOUBLE,&count);
26         printf("Received %d elements\n",count);
27     }
28     MPI_Finalize();
29     return 0;
30 }
```

3.2 Функції колективного обміну MPI

Функції колективного обміну призначені для обміну даними між усіма процесами заданої області зв'язку (комунікатора). У даному розділі розглядаються п'ять функцій, що відносяться до чотирьох груп колективного обміну:

- широкомовна розсилка даних;
- розподіл даних;
- збір даних;
- зведення і сканування даних, та деякі векторні модифікації відповідних функцій.

Розглянемо функції колективного обміну.

Функція `MPI_Bcast()` розсилає дані з буферу процесу, з номером `root`, усім іншим процесам у даному комунікаторі. Синтаксис функції наступний:

```
MPI_Bcast(void* buffer, int count,
          MPI_Datatype datatype, int root, MPI_Comm comm);
```

Функція `MPI_Gather()` виконує збирання блоків даних, що відсилаються усіма процесами у даному комунікаторі у порядку збільшення номерів процесів, в один масив даних процесу з номером `root`. Дані надсилаються з буферів, що передають (`send`), до тих, що приймають (`recv`). Синтаксис функції наступний:

```
MPI_Gather(void* sendbuf, int sendcount,
          MPI_Datatype sendtype, void* recvbuf, int recvcount,
          MPI_Datatype recvttype, int root, MPI_Comm comm);
```

Функція `MPI_Scatter()` розбиває масив даних у буфері `sendbuf` процесу з номером `root` на рівні (`sendcount`) частини і надсилає їх до інших процесів даного комунікатора (`comm`) у порядку зростання номерів процесів. Процеси, що приймають дані, розміщують їх у своїх буферах `recvbuf`. Синтаксис функції наступний:

```
MPI_Scatter(void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, int recvcount,
            MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Функція `MPI_Reduce()` здійснює операцію редукції `op` над елементами даних у буферах `sendbuf` усіх процесів у даному комунікаторі `comm` і розміщує результати редукції у буфері `recvbuf` процесу `root`. Синтаксис функції наступний:

```
MPI_Reduce(void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, int root,
            MPI_Comm comm);
```

Функція *префіксної редукції* `MPI_Scan()` надсилає до приймального буфера (`recvbuf`) *i*-го процесу результат операції `op` над значеннями з буферів передачі (`sendbuf`) процесів з номерами `0..i` включно. Синтаксис функції наступний:

```
MPI_Scan(void* sendbuf, void* recvbuf, int count,
          MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

У якості операції `op` використовується одна з визначених у бібліотеці MPI операцій (`MPI_SUM`, `MPI_PROD`, `MPI_MAX` та ін.) або операція, створена користувачем (див. лістинг 3.3).

Лістинг 3.3 – Програма з колективними функціями обміну даних

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char **argv)
5  {
6      int n=16, m;
7      int *x, *y, *z;
8      int sum = 0.0 , total = 0.0;
9      int rank, size;
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```

14
15 if(n%size != 0)
16 {
17     printf("n must be a multiple to size \n");
18     MPI_Abort(MPI_COMM_WORLD, 1);
19 }
20
21 if(rank == 0)
22 {
23     x = new int[n];
24     for(int i = 0; i < n; i++)
25         x[i] = i;
26     m = n/size;
27 }
28
29 MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
30
31 y = new int[m];
32 z = new int[m];
33
34 MPI_Scatter(x, m, MPI_INT, y, m, MPI_INT, 0,
35             MPI_COMM_WORLD);
36 for(int i = 0; i<m; i++)
37     sum += y[i];
38 MPI_Barrier(MPI_COMM_WORLD);
39 printf("%d: sum = %d\n", rank, sum);
40
41 MPI_Reduce(&sum, &total, 1, MPI_INT, MPI_SUM, 0,
42            MPI_COMM_WORLD);
43
44 if(rank == 0)
45 {
46     printf("total = %d\n", total);
47     for(int i = 0; i < m; i++)
48         printf("z[%d] = %d\n", i, z[i]);
49 }
50
51 MPI_Scan(y, z, m, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
52

```

```

53 for(int i = 0; i < size ; i++)
54     if(rank == i)
55     {
56         printf("%d: ", rank);
57         for(int j = 0; j < m; j++)
58             printf("%.d ", z[j]);
59         printf("\n");
60     }
61 MPI_Gather(z, 4, MPI_INT, x, 4, MPI_INT, 0,
62           MPI_COMM_WORLD);
63 if(rank == 0)
64 {
65     for(int i = 0; i < n; i++)
66         printf("%d ", x[i]);
67     delete[] x;
68 }
69 delete[] y;
70 delete[] z;
71
72 MPI_Finalize();
73 return 0;
74 }

```

3.3 Застосування MPI у практичних обчисленнях

3.3.1 Розподіл обчислень між процесами

Програма 3.3.1. Паралельне обчислення суми ряду

У програмі паралельно обчислюється сума ряду $\sum_{i=0}^{N-1} a_i \cdot x_i$.

Для цього до кожного з P процесів за допомогою функції `MPI_Scatter()` передаються відповідні частини масивів чисел a_i і x_i , що складаються з $M = N/P$ елементів. Процеси обчислюють часткові суми ряду. Після завершення цих обчислень загальна сума ряду визначається за допомогою функції `MPI_Reduce()`. Програмну реалізацію надано у лістингу 3.4.

Лістинг 3.4 – Програма обчислення суми ряду

```

1  #include <stdio.h>
2  #include <math.h>
3  #include <time.h>
4  #include <stdlib.h>
5  #include "mpi.h"
6  #define n 4000000
7  int main(int argc, char *argv[])
8  {
9      int p, m;
10     int rank; long i, j;
11     double sum = 0.0, total;
12     double t1, t2;
13     double* sx;
14     double* sa;
15     double* rx = new double[n];
16     double* ra = new double[n];
17     if(rx == NULL || ra == NULL)
18     {
19         printf("no memory allocated for arrays rx or ra");
20         exit(1);
21     }
22
23     MPI_Init(&argc,&argv);
24     MPI_Comm_size(MPI_COMM_WORLD,&p);
25     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
26
27     if(rank==0)
28     {
29         srand((unsigned)time(NULL));
30         sx = new double[n];
31         sa = new double[n];
32         if(sx == NULL || sa == NULL)
33         {
34             printf("no memory allocated for
35                     arrays sx or sa");
36             MPI_Abort(MPI_COMM_WORLD, 1);
37         }
38         for(i=0;i<n;i++)
39         {

```

```

40         sx[i]= rand()*1E-8;
41         sa[i]= rand()*1E-8;
42     }
43     m=n/p;
44 }
45 MPI_Bcast(&m,1,MPI_INT,0,MPI_COMM_WORLD);
46 MPI_Scatter(sx,m, MPI_DOUBLE,rx,m,MPI_DOUBLE,0,
    MPI_COMM_WORLD);
47 MPI_Scatter(sa,m,MPI_DOUBLE,ra,m,MPI_DOUBLE,0,
    MPI_COMM_WORLD);
48 if(rank == 0)
49 {
50     t1= MPI_Wtime();
51 }
52 for(i=0;i<m;i++) sum += ra[i]*rx[i];
53 MPI_Barrier(MPI_COMM_WORLD);
54 MPI_Reduce(&sum,&total,1,MPI_DOUBLE,MPI_SUM,0,
    MPI_COMM_WORLD);
55 if(rank == 0)
56 {
57     t2= MPI_Wtime() - t1;
58     printf("t =%.5f sec.\n",t2);
59     printf("sum in %d procs is %.12f\n",p, total);
60     delete[] sx;
61     delete[] sa;
62 }
63
64 delete[] rx;
65 delete[] ra;
66
67 MPI_Finalize();
68 return 0;
69 }

```

Програма 3.3.2. Приклад рівномірного розподілу навантаження між процесами

Нехай необхідно обчислити суму ряду $S = \sum_{i=1}^N a_i \cdot x_i^i$, де N достатньо велике число, і обчислення слід розподілити між P процесами, причому ступені чисел x_i мають обчислюватися шляхом їх перемноження.

Якщо у кожному процесі обчислювати частину суми, що складається з N/P елементів вихідного ряду, то це призведе до нерівномірного завантаження процесів. Наприклад, якщо $N=30$, $P=3$, то розбиття ряду на три частини по N/P елементів дасть наступне:

$$S = \sum_{i=1}^{10} a_i \cdot x_i^i + \sum_{i=11}^{20} a_i \cdot x_i^i + \sum_{i=21}^{30} a_i \cdot x_i^i.$$

У результаті цього перший процес виконає 55, другий – 155, а третій – 255 перемножень.

У даній програмі реалізовано приблизно рівномірний розподіл обсягу обчислень цієї модельної задачі між процесами наступним чином. У другому елементі розглянутого ряду необхідно виконати одне перемноження, у третьому – два і т.д. Загальна кількість перемножень для будь-якого «відрізка» ряду з $M = N/P$ елементів формує арифметичну прогресію, тобто маємо наступне:

$$S_M = n \cdot M + \frac{(M-1) \cdot M}{2}, \quad (3.1)$$

де n – номер першого елемента відрізка.

Тоді число операцій перемноження, які має виконати кожен процес, можна визначити за наступною формулою:

$$S_N = \frac{1}{P} \cdot \left(N + \frac{(N-1) \cdot N}{2} \right) = \frac{N \cdot (N+1)}{2 \cdot P}. \quad (3.2)$$

Для довжини відрізка ряду, який має оброблятися першим процесом при рівномірному розподілі числа перемножень між усіма процесами, складемо наступне рівняння:

$$n_1 \cdot M_1 + \frac{(M_1-1) \cdot M_1}{2} = \frac{N \cdot (N+1)}{2 \cdot P}, \quad (3.3)$$

де $n_1 = 1$.

Це квадратне рівняння відносно невідомої змінної M_1 , яке можна представити наступним чином:

$$P \cdot M_1^2 + P \cdot (2 \cdot n_1 - 1) \cdot M_1 - N \cdot (N + 1) = 0. \quad (3.4)$$

Розв'язуючи рівняння (3.4), знайдемо значення M_1 – число елементів першого відрізка.

Початком другого відрізка, очевидно, буде елемент з номером $n_2 = n_1 + M_1$, тоді число елементів другого відрізка, призначеного для обробки другим процесом, можна знайти, склавши і розв'язавши наступне рівняння:

$$n_2 \cdot M_2 + \frac{(M_2 - 1) \cdot M_2}{2} = \frac{N \cdot (N + 1)}{2 \cdot P}, \quad n_2 = n_1 + M_1. \quad (3.5)$$

Аналогічно можна знайти число елементів для усіх відрізків.

Для реалізації нерівномірного розподілу елементів масиву між процесами можна використати векторну версію колективної функції розподілу, яка має наступний синтаксис:

```
MPI_Scatterv(void* sendbuf, int *sendcounts,
             int *offsets, MPI_Datatype sendtype,
             void* recvbuf, int recvcnt, MPI_Datatype recvtpe,
             int root, MPI_Comm comm);
```

де `sendcounts` – масив цілих чисел (розмір масиву дорівнює числу процесів у групі), що містить числа – кількість елементів, які надсилаються кожному процесу;

`offsets` – масив цілих чисел (розмір масиву дорівнює числу процесів у групі), в якому i -е значення визначає зміщення відносно початку `sendbuf` для даних, що надсилаються i -му процесу.

У програмі лістингу 3.5 функція `offset()` призначена для розв'язання заданого квадратного рівняння. У рядках 39–46 даного лістингу обчислюються зміщення ділянок масиву відносно його початку (масив `shift`) і число елементів у кожній ділянці (масив `scounts`).

Лістинг 3.5 – Програма з векторною колективною функцією розподілу

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <time.h>
4  #include <stdlib.h>
5  #include "mpi.h"
6
7  #define n 4500
8
9  int offset(int i, int a, int N)
10 {
11     int x;
12     int b = (2*i -1)*a;
13     int c = -N*(N+1);
14     x = int(-b + sqrt(b*b -4*a*c))/(2.0* a);
15     return x;
16 }
17
18 int main(int argc, char *argv[])
19 {
20     double* sx;
21     double* sa;
22     double* rx;
23     double* ra;
24
25     int p;
26     int rank;
27     double sum = 0., total = 0., mul;
28     double t1, t2;
29
30     MPI_Init(&argc,&argv);
31     MPI_Comm_size(MPI_COMM_WORLD,&p);
32     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
33
34     int* shift = new int[p];
35     int* scounts = new int[p];
36
37     if(rank ==0)
38     {
39         shift[0] = 0;
```

```

40     int k =0;
41     for(int i = 0; i < p; i++)
42     {
43         shift [i] = k;
44         k = k + offset(k, p, n);
45         scounts[i] = k - shift[i];
46     }
47     for(int i = 0; i < p; i++)
48     {
49         printf("%d ", shift[i]);
50     }
51     printf("\n");
52     for(int i = 0; i < p; i++)
53     {
54         printf("%d ", scounts[i]);
55     }
56     printf("\n");
57     srand(time(0));
58     sx = new double[n];
59     sa = new double[n];
60     if(sx == NULL || sa == NULL)
61     {
62         printf("not memory allocated for arrays sx or sa");
63         MPI_Abort(MPI_COMM_WORLD, 1);
64     }
65     for(int i = 0; i < n; i++)
66     {
67         sx[i] = 0.9+0.1*double(rand())/RAND_MAX;
68         sa[i] = double(rand())/RAND_MAX;
69     }
70 }
71
72 MPI_Bcast(shift,p, MPI_INT,0, MPI_COMM_WORLD);
73 MPI_Bcast(scount,p, MPI_INT,0, MPI_COMM_WORLD);
74
75 rx = new double[scount[0]];
76 ra = new double[scount[0]];
77
78 MPI_Scatterv(sx,scount,shift,MPI_DOUBLE,rx,
              scount[0],MPI_DOUBLE,0,MPI_COMM_WORLD);
79 MPI_Scatterv(sa,scount,shift,MPI_DOUBLE,ra,
              scount[0],MPI_DOUBLE,0,MPI_COMM_WORLD);

```

```

80
81 if(rank == 0)
82 {
83     t1 = MPI_Wtime();
84 }
85 for(int i = 0; i < scounts[rank]; i++)
86 {
87     mul = 1.0;
88     for(int j = 0; j < i + shift[rank]; j++)
89     {
90         mul *= rx[i];
91     }
92     sum += ra[i]*mul;
93 }
94
95 MPI_Barrier(MPI_COMM_WORLD);
96 if(rank == 0)
97 {
98     t2 = MPI_Wtime() - t1;
99 }
100
101 MPI_Reduce(&sum,&total,1,MPI_DOUBLE,MPI_SUM,0,
102           MPI_COMM_WORLD);
103
104 if(rank == 0)
105 {
106     printf("total = %.12f\n",total);
107     printf("time_parallel = %.4f\n",t2);
108 }
109
110 delete[] sx;
111 delete[] sa;
112
113 delete[] shift;
114 delete[] scounts;
115
116 delete[] rx;
117 delete[] ra;
118
119 MPI_Finalize();
120 return 0;
121 }

```

3.3.2 Паралельні обчислення за допомогою методу Монте-Карло

При інтегруванні складних функцій і за складної форми області інтегрування можна застосувати метод Монте-Карло.

Розглянемо ідею метода Монте-Карло на простому прикладі.

Площа кола S_C з радіусом r і площа описаного навколо нього квадрата S_R обчислюються за наступними формулами:

$$S_C = \pi \cdot r^2, \quad S_R = 4 \cdot r^2. \quad (3.6)$$

За допомогою генератора випадкових чисел будемо генерувати координати точок, рівномірно розподілених всередині квадрата R , одночасно підраховуючи кількість точок, що потрапили до кола (рис. 3.1).

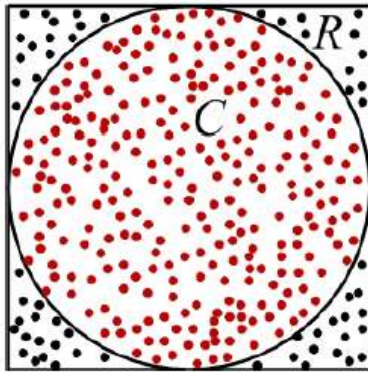


Рисунок 3.1 – Розподіл координат для задачі обчислення числа π

Нехай кількість згенерованих точок дорівнює n , а кількість точок, що потрапили до кола – m ($m \leq n$). Тоді, за достатньо великого значення n , відношення m до n буде пропорційним відношенню площі кола S_C до площі квадрата S_R (у силу рівномірності розподілу точок), тобто матимемо наступне:

$$\lim_{n \rightarrow \infty} \frac{m}{n} \approx \frac{S_C}{S_R} = \frac{\pi \cdot r^2}{4 \cdot r^2} = \frac{\pi}{4}, \quad (3.7)$$

а отже

$$\pi \approx \lim_{n \rightarrow \infty} \frac{4 \cdot m}{n}. \quad (3.8)$$

Одержуване значення числа π є випадковим. Виконаємо серію з N незалежних реалізацій обчислення цього числа таким способом. Позначимо одержувані значення π як $\xi_1, \xi_2, \dots, \xi_N$, а їх середнє – наступним чином:

$$\bar{\xi}_N = \frac{\xi_1 + \xi_2 + \dots + \xi_N}{N}. \quad (3.9)$$

За достатньо великого значення N для похибки середнього обчисленого значення числа π у силу центральної граничної теореми і згідно відомого правила «трьох сигм» можна записати наступний вираз:

$$p\left(\left|\pi - \bar{\xi}_N\right| < 3 \cdot \frac{\sigma}{\sqrt{N}}\right) \approx 0,997, \quad (3.10)$$

тобто нерівність $\left|\pi - \bar{\xi}_N\right| < 3 \cdot \frac{\sigma}{\sqrt{N}}$ виконується з вірогідністю, достатньо близькою до одиниці ($p \approx 0,997$), де σ^2 – дисперсія випадкової величини ξ_i , $i = \overline{1, N}$.

Розглянемо використання методу Монте-Карло для інтегрування. Для цього візьмемо інтеграл

$$I = \int_0^1 f(x) dx. \quad (3.11)$$

Нехай η – рівномірно розподілена на відрізку $[0, 1]$ випадкова величина, тобто її щільність розподілу ймовірностей визначається наступним:

$$p_{\eta}(x) = \begin{cases} 1, & \text{якщо } x \in [0, 1]; \\ 0, & \text{якщо } x \notin [0, 1]. \end{cases} \quad (3.12)$$

Тоді $\xi = f(\eta)$ теж буде деякою випадковою величиною, причому її математичне очікування

$$M[\xi] = \int_0^1 f(x) \cdot p_{\eta}(x) dx = \int_0^1 f(x) dx = I. \quad (3.13)$$

Таким чином,

$$I = \int_0^1 f(x) dx \approx \bar{\xi}_N = \frac{1}{N} \cdot \sum_{i=1}^N f(\eta_i), \quad (3.14)$$

де η_i – незалежні реалізації рівномірно розподіленої на відрізку $[0, 1]$ випадкової величини η .

Якщо ε – задана допустима похибка обчислення інтеграла, то, згідно (3.10), з вірогідністю, близькою до одиниці, матимемо наступне:

$$\varepsilon = 3 \cdot \frac{\sigma}{\sqrt{N}}. \quad (3.15)$$

Кратні інтеграли обчислюються аналогічним чином:

$$I = \int_0^1 \int_0^1 f(x, y) dx dy = \frac{1}{N} \cdot \sum_{i=1}^N f(\eta_i, \theta_i), \quad (3.16)$$

де η_i, θ_i – незалежні реалізації рівномірно розподілених на відрізьку $[0, 1]$ випадкових величин η і θ .

Інтегрування методом Монте-Карло також зручне, якщо необхідно обчислити кратний інтеграл по деякій області Ω , що має складну форму (3.14). У такому випадку припускають наступне:

$$I = \iint_{\Omega} f(P) dP = \iint_R f^*(P) dP, \quad (3.17)$$

де R – площа квадрата, у який вписано задану область Ω :

$$f^*(P) = \begin{cases} f(P), & \text{якщо } P \in \Omega; \\ 0, & \text{якщо } P \notin \Omega. \end{cases} \quad (3.18)$$

Далі використовують формулу (3.16).

Рішення задач методом Монте-Карло добре розпаралелюється, оскільки генерувати випадкові числа і обчислювати від них функції можна незалежно у різних процесах. При цьому, як правило, загальне число необхідних обчислень (генерування випадкових чисел, обчислення функцій і складання їх значень) рівномірно розподіляється між процесами.

Програма 3.3.3. Паралельне обчислення кратного інтеграла

Нехай необхідно обчислити подвійний інтеграл

$$\iint_{\Omega} f(x \cdot y - 4 \cdot x + 2 \cdot y - 1) dx dy, \quad (3.19)$$

де область інтегрування Ω задана рівняннями $x=1, y=x^2, y=0$ і показана на рис. 3.2.

Якщо генерувати випадковим чином розподілені у прямокутнику R координати, то частина з них буде належати області Ω . Тоді, складаючи значення функції $f(x, y)$ у координатах, що належать до області Ω , і розділивши їх на загальну кількість згенерованих координат точок в області квадрата, за формулою (3.16) одержимо оцінку шуканого інтеграла.

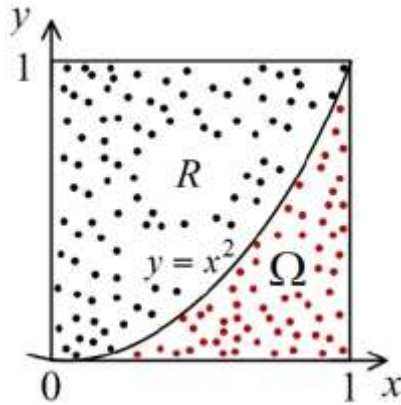


Рисунок 3.2 – Визначення області інтегрування кратного інтеграла

Програму, що реалізує метод обчислення кратного інтеграла, наведено у лістингу 3.6. Для демонстрації працездатності і оцінки точності обчислення названого інтеграла у програмі порівнюється обчислене значення інтеграла з його аналітичним рішенням $I = -21/20$ (змінна *analytical*).

Зверніть увагу на організацію генерування випадкових значень координат. Для того, щоб кожний процес генерував власний набір випадкових чисел, у функцію `srand()` у якості аргументу передається змінна `seed` (рядки 23–25), так зване «зерно», що ініціалізує генератор псевдовипадкових чисел. Змінна `seed` враховує поточне значення часу у мікросекундах і ранг процесу.

Кожен процес обчислює власну суму значень функції від координат (змінна `f`) і за допомогою функції `MPI_Reduce()` визначається результуюча сума усіх реалізацій (змінна `ft`).

Лістинг 3.6 – Програма обчислення кратного інтеграла

```

1  #include "mpi.h"
2  #include <stdio.h>
3  #include <math.h>
4  #include <sys/time.h>
5  #include <stdlib.h>
6  #define n 500000
7

```

```

8  int main(int argc, char *argv[])
9  {
10 int rank, p;
11 double analytical = -21.0/20;
12 double f = 0.0;
13 double ft;
14 double x, y;
15
16 int seed;
17 timeval tv;
18
19 MPI_Init(&argc,&argv);
20 MPI_Comm_size(MPI_COMM_WORLD,&p);
21 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
22
23 gettimeofday(&tv,NULL);
24 seed = tv.tv_usec*(rank+1);
25 srand(seed);
26
27 for(long i=0; i<n; i++)
28 {
29     x = double(rand())/RAND_MAX;
30     y = double(rand())/RAND_MAX;
31
32     if(y <= x*x)
33     {
34         f += (x*y - 4*x + 2*y - 1.0) /n;
35     }
36 }
37
38 MPI_Reduce(&f,&ft,1,MPI_DOUBLE,MPI_SUM,0,
39           MPI_COMM_WORLD);
40
41 if(rank ==0)
42 {
43     ft = ft/p;
44     printf("ft =%.16f; err =%.16f, points =%d\n", ft,
45           fabs(analytical - ft),n*p);
46 }
47 MPI_Finalize();
48 return 0;
49 }

```

3.4 Завдання до лабораторної роботи

3.4.1 Завдання до розділів 3.1 і 3.2. Реалізувати і вивчити програми, надані у лістингах 3.1–3.3. Пояснити результати роботи програм.

3.4.2 Завдання до програми 3.3.1. Реалізувати програму, надану у лістингу 3.4 та на основі цієї програми розробити власну програму згідно варіанту.

Зауваження: варіант визначається як залишок від ділення номеру робочого місця на 2: 1 – 1-ий варіант; 0 – 2-ий.

Варіант 1. Програма має обчислювати суму ряду $\sum_{i=0}^{N-1} a_i \cdot \sin(x_i)$, де $N = 4 \cdot 10^6$, $x_i = i \cdot 10^{-4}$, a_i – випадкові числа у діапазоні $(-1, 1)$. Функція $\sin(x_i)$ обчислюється за допомогою розкладу у ряд:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^n \cdot \frac{x^{2n+1}}{(2 \cdot n + 1)!} + \dots \quad (3.20)$$

Для обчислення кожного значення функції $\sin(x_i)$ розробити функцію, що працює згідно відповідного алгоритму, наданого у табл. 3.1.

Варіант 2. Програма має обчислювати суму ряду $\sum_{i=0}^{N-1} a_i \cdot \cos(x_i)$, де $N = 4 \cdot 10^6$, $x_i = i \cdot 10^{-4}$, a_i – випадкові числа у діапазоні $(-1, 1)$. Функція $\cos(x_i)$ обчислюється за допомогою розкладу у ряд:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots + (-1)^n \cdot \frac{x^{2n}}{(2 \cdot n)!} + \dots \quad (3.21)$$

Для обчислення кожного значення функції $\cos(x_i)$ розробити функцію, що працює згідно відповідного алгоритму, наданого у табл. 3.1.

Таблиця 3.1 – Алгоритми для програмної реалізації функції згідно варіанту

Варіант 1 (3.20)	Варіант 2 (3.21)
<i>input: x</i> <i>output: sin x</i> $y \leftarrow x$ $s \leftarrow y$ <i>for k</i> $\leftarrow 1$ <i>to 50 do</i> $y \leftarrow -\frac{x^2}{(k+1) \cdot (k+2)} \cdot y$ $s \leftarrow s + y$ $k \leftarrow k + 2$ <i>end</i> $\sin x \leftarrow s$	<i>input: x</i> <i>output: cos x</i> $y \leftarrow 1$ $c \leftarrow y$ <i>for k</i> $\leftarrow 0$ <i>to 50 do</i> $y \leftarrow -\frac{x^2}{(k+1) \cdot (k+2)} \cdot y$ $c \leftarrow c + y$ $k \leftarrow k + 2$ <i>end</i> $\cos x \leftarrow c$

3.4.3 Завдання до програми 3.3.2 (лістинг 3.5).

3.4.3.1 Вивчити і реалізувати програму.

3.4.3.2 Допрацювати код програми таким чином, щоб у кінці програми у нульовому процесі обчислювалася уся сума ряду для перевірки правильності паралельної реалізації обчислення.

3.4.3.3 Провести достатню кількість експериментів, запускаючи програму для двох, трьох і чотирьох процесів на одній робочій станції, і визначити максимальне прискорення для цих випадків. Пояснити одержувані результати.

3.4.3.4 Провести достатню кількість експериментів, запускаючи програму для двох, трьох і чотирьох робочих станцій, і визначити максимальне прискорення для цих випадків. Пояснити одержувані результати.

3.4.3.5 Додати до програми код, за допомогою якого можна обчислити кількість операцій множення, що виконуються у кожному

процесі, та переконатися у тому, що процеси завантажені приблизно однаково.

3.4.4 Завдання до програми 3.3.3 (лістинг 3.6).

3.4.4.1 Вивчити і реалізувати програму.

3.4.4.2 Дати відповідь на питання – у даній програмі зі зростанням числа процесорів збільшується прискорення чи точність рішення?

3.4.4.3 Визначити, у скільки разів потрібно збільшити число генерованих координат, щоб допустима похибка обчислення інтегралу зменшилась приблизно на порядок. Відповідь обґрунтувати теоретично і підтвердити експериментально.

3.4.4.4 Побудувати графік залежності похибки обчислення кратного інтегралу від числа генерованих координат, провівши достатню кількість експериментів.

3.4.4.5 Доповнити програму кодом для виміру часу виконання програми.

3.4.4.6 Провести достатню кількість запусків програми на чотирьох паралельно працюючих станціях, побудувати графік залежності часу виконання програми від числа згенерованих координат.

3.4.4.7 Розробити програму згідно варіанту:

– програма має обчислювати кратний інтеграл

$$\iint_{\Omega} xy \, dx dy. \quad (3.22)$$

Варіант 1. Область інтегрування Ω визначається співвідношеннями $0 \leq x \leq 1$, $x^2 \leq y(x) \leq \sqrt{x}$.

Варіант 2. Область інтегрування – чверть кола з центром у нульовій точці координат і радіусом $r = 0,5$ і визначається співвідношеннями $x^2 + y^2 \leq r^2$, $x \geq 0$, $y \geq 0$.

3.4.4.8 Довести правильність чисельного розв’язання з використанням розробленої програми, провівши порівняння одержаного результату з аналітичним розв’язанням (знайти самостійно, наприклад, в Інтернеті, за допомогою пакету *Scilab*, іншого чисельного методу чи розв’язавши аналітично).

3.4.4.9 Виконати завдання п.п.3.4.4.4–3.4.4.6 для розробленої програми.

3.4.4.10 Результати експериментів та їх інтерпретацію навести у звіті.

3.5 Зміст звіту

3.5.1 Титульний лист.

3.5.2 Мета роботи.

3.5.3 Фрагменти програмного коду, розроблені самостійно.

3.5.4 Графіки, побудовані згідно завдань.

3.5.5 Результати проведених експериментів і їх інтерпретація.

3.5.6 Відповіді на контрольні питання. Вміти їх пояснювати в усній формі.

3.6 Контрольні питання

3.6.1 Чим відрізняються функції колективного обміну від функцій двохточкового обміну MPI?

3.6.2 У чому відмінність блокуючих і неблокуючих функцій обміну MPI?

3.6.3 Описати функції `MPI_Sendrecv()` і `MPI_Sendrecv_replace()`.

3.6.4 Описати функцію `MPI_Scan()`.

3.6.5 Навести описи функцій `MPI_Scatterv()`, `MPI_Gatherv()`, `MPI_Alltoall()`, `MPI_Alltoallv()`, `MPI_Allgather()` і `MPI_Allgatherv()`.

3.6.6 Описати конструктори груп і комунікаторів.

ЛАБОРАТОРНА РОБОТА №4

ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ OPENMP

Мета роботи – вивчити можливості технології OpenMP для паралельного багатопоточного програмування.

4.1 Основні положення і поняття технології OpenMP

OpenMP – стандарт, що визначає набір директив компілятора, бібліотечних функцій і змінних середовища оточення, для створення багатопоточних програм, а також паралельних програм за допомогою багатопоточності [7].

Програма з використанням OpenMP – це сукупність послідовних фрагментів коду і паралельних регіонів. Потоки, створювані у програмі, мають логічні номери, що починаються з нуля. Головний потік має номер 0. Паралельні регіони мають бути вкладеними.

Організація обчислень у технології OpenMP здійснюється за допомогою *директив* компілятора, які починаються з ключового слова `omp`. В OpenMP передбачено три категорії директив:

- визначення паралельної області;
- розподілу обчислень;
- синхронізації.

Директива – це директива компілятору чи прагма (препроцесорна директива) мови C/C++. Директиви зазвичай застосовуються по відношенню до структурних блоків (див. лістинг 4.1).

Лістинг 4.1 – Шаблон застосування директиви

```

1  #pragma omp directive[clause[clause]]...
2  {
3      structured block
4  }
```

Структурний блок – це оператор чи послідовність операторів з однією точкою входу і однією точкою виходу. Опції (`clause`) – це модифікатори директиви, що впливають на її поведінку.

Компілятор з підтримкою OpenMP визначає макрос `_OPENMP`, що може використовуватися для компіляції окремих блоків паралельної програми з умовами. Цей макрос визначено у форматі `yyyymm`, де `yyyy` і `mm` – цифри року і місяця прийняття стандарту OpenMP. За допомогою наступної програми можна перевірити, чи підтримується компілятором бібліотека OpenMP (лістинг 4.2) [4].

Лістинг 4.2 – Перевірка підтримки OpenMP компілятором

```

1  #include <stdio.h>
2  int main()
3  {
4  #ifdef _OPENMP
5      printf("OpenMP %d is supported\n",_OPENMP);
6  #endif
7      return 0;
8  }
```

Результат роботи програми, наведеної у лістингу 4.2, може бути наступним:

OpenMP 201107 is supported.

Таке повідомлення означає, що компілятор підтримує бібліотеку OpenMP від 2011 р., тобто версії 3.1.

За замовчуванням програми на OpenMP працюють у динамічному режимі, при якому кількість потоків визначається операційною системою.

Задати явно число потоків можна:

– за допомогою директиви `#pragma omp parallel num_threads(n)`, де `n` – число потоків;

– за допомогою функції `omp_set_num_threads(n)`.

Наступні функції і директиви часто використовуються для керування змінними середовища:

– `omp_get_thread_num()` – повертає номер потоку;

– `omp_get_num_threads()` – повертає число потоків у паралельній області;

– `omp_get_max_threads()` – повертає максимально допустиме число потоків у наступній паралельній області;

`omp_get_num_procs()` – повертає кількість доступних процесорів;

`omp_set_nested(int f)` – дозволяє ($f=1$) чи забороняє ($f=0$) вкладений паралелізм;

`#pragma omp single` – директива задає ділянку коду у паралельній області, яка має виконуватися лише один раз;

`#pragma omp master` – директива задає ділянку коду, яка має виконуватися лише головним потоком.

Змінні у паралельних регіонах діляться на два класи:

`shared` – загальні для всіх потоків. Загальні змінні існують в одному екземплярі і доступні для всіх потоків;

`private` – локальні для кожного потоку. Визначення локальної змінної породжує її власний екземпляр у кожному потоці.

Слід пам'ятати, що при записі потоками значень до загальної змінної може виникнути конфлікт – «гонка даних» (data race). За замовчуванням усі змінні, які визначені поза паралельним регіоном, є загальними, окрім змінних – лічильників ітерацій циклів. Рекомендується визначати клас змінних явно – за допомогою опцій `shared` і `private`. У наступному прикладі демонструється визначення загальних і локальних змінних (див. лістинг 4.3).

Лістинг 4.3 – Загальні і локальні дані

```

1  #include <stdio.h>
2  #include <omp.h>
3  #define n 10
4  int main()
5  {
6  int a[n],i;
7  #pragma omp parallel num_threads(n) shared(a)
   private(i)
8  {
9      i = omp_get_thread_num();
10     a[i] = i;
11 }
12 for(i = 0; i < n; i++)
13     printf ("%d ", a[i]);
14 return 0;
15 }
```

В OpenMP існує декілька способів розподілу обчислень між потоками:

- низькорівневий розподіл з використанням функцій `omp_get_thread_num()` і `omp_get_num_threads()`. Цей підхід аналогічний до організації паралельних програм за допомогою функцій MPI: `MPI_Comm_rank()` та `MPI_Comm_size()`;
- розподіл ітерацій циклу за допомогою директиви `for`;
- шляхом розбиття задачі на незалежні секції з використанням директиви `section`;
- шляхом виділення структурного блоку у якості задачі у поточному потоці (директива `task`).

На прикладі лістингу 4.4 показано низькорівневий розподіл обчислень.

Лістинг 4.4 – Низькорівневий розподіл обчислень

```

1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5      omp_set_num_threads(2);
6      #pragma omp parallel
7      {
8          int lpt = omp_get_num_threads();
9          lpt = 26/lpt; // символів на потік
10         int m = omp_get_thread_num();
11         int sl = 'a' + m * lpt; // початковий символ
12         int el = 'a' + m * lpt + lpt; // кінцевий символ
13         for( int i=sl; i<el; i++)
14             printf ("%d: %c\n", m, i);
15     }
16     return 0;
17 }
```

У програмі, наведеній у лістингу 4.5, реалізовано обчислення добутку двох матриць. Розподіл обчислень за ітераціями між трьома потоками здійснюється за допомогою директиви `for`.

Лістинг 4.5 – Розподіл обчислень за ітераціями

```

1  #include <stdio.h>
2  #include <omp.h>
3  #define n 3
4
5  int main()
6  {
7      int m,i,j,k;
8      int a[n][n] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
9      int b[n][n] = {{2, 1, 1}, {4, 5, 3}, {3, 1, 2}};
10     int c[n][n] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
11     omp_set_num_threads(3);
12     #pragma omp parallel shared(a,b,c) private(i,j,k)
13     {
14         #pragma omp for
15         for(i=0;i<n;i++)
16             for(j=0;j<n;j++)
17             {
18                 for(k=0;k<n;k++)
19                     c[i][j] += a[i][k]*b[k][j];
20                 m = omp_get_thread_num();
21                 printf("t%d: c[%d][%d] = %d\n",m,i,j,c[i][j]);
22             }
23     }
24     for(i=0;i<n;i++)
25     {
26         for(j=0;j<n;j++)
27             printf ("%d ",c[i][j]);
28         printf("\n");
29     }
30     return 0;
31 }

```

У наступному прикладі (див. лістинг 4.6) застосовано опцію «редукції» `reduction(operator:list)`, що виконує заданий оператор `operator` над локальними змінними `list` кожного з потоків. Оператор – це одна з операцій мови C: `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`. У цьому ж прикладі використано функцію `OpenMP` `omp_get_wtime()` для одержання значення поточного часу.

Програма обчислює наближене значення числа π за відомою формулою:

$$\int_0^1 \frac{4}{1+x^2} dx = 4 \cdot \arctan(x) \Big|_0^1 = \pi. \quad (4.1)$$

Інтеграл (4.1) апроксимується за допомогою наступного виразу:

$$\pi \approx h \sum_{i=0}^{n-1} \frac{4}{1+x_i^2}, h = \frac{1}{n}, x_i = \left(i + \frac{1}{2}\right) \cdot h, \quad (4.2)$$

де h – крок ітерації, n – число ітерацій.

У програмі (див. лістинг 4.6) реалізовано паралельне (рядки 12–20) і послідовне (рядки 27–32) обчислення значення π , і визначення прискорення як відношення часу послідовного обчислення до часу паралельного обчислення.

Лістинг 4.6 – Обчислення значення π

```

1  #include <stdio.h>
2  #include <omp.h>
3
4  double f( double y) {return (4.0/(1.0+ y*y));}
5  int main()
6  {
7      double h, x, pi, t1, t2, S, sum = 0.0, ts , tp;
8      int i, n = 50000000;
9      h = 1.0/n;
10 //omp_set_num_threads(4);
11     tp= omp_get_wtime();
12 #pragma omp parallel private(x,i)
13     {
14         #pragma omp for reduction(+: sum)
15         for(i=0; i<n; i++)
16         {
17             x = h*(i + 0.5);
18             sum = sum + f(x);
19         }

```

```

20     }
21     pi = h*sum;
22     tp= omp_get_wtime() - tp;
23     printf("n = %d, pi = %12.11f, tp = %f\n",
           n,pi,tp);
24 // --- послідовне обчислення -----
25     sum =0.0;
26     ts= omp_get_wtime();
27     for (i=0; i<n; i++)
28     {
29         x = h*(i +0.5);
30         sum = sum + f(x);
31     }
32     pi = h*sum;
33     ts= omp_get_wtime() - ts;
34     S = ts/tp;
35     printf("n = %d, pi = %12.11f, ts = %f, S = %f\n",
           n,pi ,ts ,S);
36     return 0;
37 }

```

4.2 Застосування OpenMP для практичних обчислень

Сортування неупорядкованих масивів (символів, рядків, чисел) є складовою частиною багатьох програм. Обчислювальна складність сортування залежить від розміру масивів, а часові витрати на сортування можуть складати значну частину загального часу роботи програми. У зв'язку із цим для сортування буває доцільно використовувати паралельні методи, зокрема з використанням розпаралелювання на потоки, які виконуються на окремих ядрах чи процесорах. Існує багато алгоритмів сортування, що відрізняються за своєю ефективністю. У даному розділі ми розглянемо послідовні і паралельні реалізації декількох методів сортування.

4.2.1 Бульбашкове послідовне сортування

Спочатку розглянемо звичайний алгоритм бульбашкового сортування. Основною операцією алгоритмів сортування є операція «порівняти-обміняти». Послідовний алгоритм «розглядає» спочатку перший елемент масиву, порівнює його із сусіднім елементом і,

за необхідності, обмінює їх місцями. Так алгоритм проходить у результаті першої ітерації усі елементи масиву, і найбільший елемент масиву виявляється першим. Цей процес схожий на «спливання» бульбашки. Потім алгоритм працює з другим елементом одержаного масиву і виконує з ним таку саму процедуру. Після $n-1$ ітерації масив гарантовано буде відсортовано за зменшенням. Ми наводимо програму, в якій реалізовано бульбашкове сортування, щоб навчити оцінювати трудомісткість алгоритмів сортування та ефективність розпаралелювання алгоритмів сортування.

Найпростіша програма сортування наведена у лістингу 4.7. Додаткова функція `fill()` призначена для ініціалізації масиву випадковими числами, а функція `check()` – для перевірки впорядкованості масиву.

Лістинг 4.7 – Програма послідовного бульбашкового сортування

```

1  #include <iostream>
2  #include <ctime>
3  #include <stdlib.h>
4
5  using namespace std;
6  void bubblesort(double*,int);
7  void fill(double*,int);
8  void check(double*,int);
9
10 int main()
11 {
12     srand(time(0));
13     int n = 10000;
14     double* a = new double[n];
15     double t;
16
17     printf("size of array n = %d\n",n);
18     fill(a,n);
19     t = double(clock());
20     bubblesort(a,n);
21     t = (double(clock())-t)/1000;
22     check(a,n);
23     printf("bubblesort time = %f sec\n",t);
24     delete[] a;
```

```

25 return 0;
26 }
27
28 void bubblesort(double* a, int n)
29 {
30     double t;
31     for(int i = 0; i < n-1; i++)
32         for(int j = 1; j < n-i; j++)
33             if(a[j-1] > a[j])
34                 {
35                     t = a[j];
36                     a[j] = a[j-1];
37                     a[j-1] = t;
38                 }
39 }
40
41 void fill(double* a, int n)
42 {
43     for(int i = 0; i < n; i++)
44         a[i] = double(rand())/RAND_MAX;
45 }
46
47 void check(double* a, int n)
48 {
49     for(int i = 1; i < n; ++i)
50         if(a[i] < a[i-1])
51             printf("error\n");
52 }

```

За допомогою наведеної програми (лістинг 4.7) можна оцінити час однієї операції порівняння-обміну. При цьому необхідно забезпечити такий режим роботи програми, за якого вибірка даних з пам'яті у кеш не відбувається. Це досягається підбором достатньо малого розміру сортованого масиву, щоб він увесь поміщався до кешу пам'яті.

Оцінку часу виконання бульбашкового сортування проведемо згідно наступних міркувань. На першій ітерації кількість операцій порівняння-обміну рівна $n-1$, на другій – $n-2$ і т.д. Отже, загальний час виконання операцій порівняння-обміну складає

$$T_c = \sum_{i=1}^{n-1} (n-i) \cdot \tau = \frac{n^2 - n}{2} \cdot \tau. \quad (4.3)$$

Якщо ширину шини пам'яті процесора позначити через ω , а її пропускну спроможність – через β , то оцінка часових витрат на переміщення даних з пам'яті до кешу, з урахуванням латентності α , може бути наступною:

$$T_m = \frac{n^2 - n}{2} \cdot \left(\alpha + \frac{\omega}{\beta} \right). \quad (4.4)$$

Для більш повної оцінки часових витрат на обмін з пам'яттю можна врахувати частоту кеш-промахів γ :

$$T_m = \frac{n^2 - n}{2} \cdot \left(\alpha + \frac{\omega}{\beta} \right) \cdot \gamma. \quad (4.5)$$

Отже, загальні часові витрати на бульбашкове сортування можна оцінити наступним чином:

$$T_B = T_c + T_m = \frac{n^2 - n}{2} \cdot \tau + \frac{n^2 - n}{2} \cdot \left(\alpha + \frac{\omega}{\beta} \right) \cdot \gamma. \quad (4.6)$$

Оцінка фізичних характеристик комп'ютера, на якому виконувалася дана програма, що реалізує алгоритм бульбашкового сортування, дала наступні параметри: $\alpha \approx 74,2$ нс, $\omega = 64$, $\beta = 12,8$ Гб/с. Значення $\tau \approx 27,0$ нс одержано шляхом статистичної обробки результатів замірів часу роботи операції порівняння-обміну при довжині сортованого масиву у 10^4 елементів. У якості значення частоти кеш-промахів для даного алгоритму було взято величину $\gamma \approx 0,0037$ з [5].

Результати модельної (за формулою (4.6)) і експериментальної оцінки залежності швидкості послідовного бульбашкового алгоритму

сортування від розміру масиву на 4-ядерному процесорі AMD A6-3650 2,6 GHz надано на рис. 4.1.

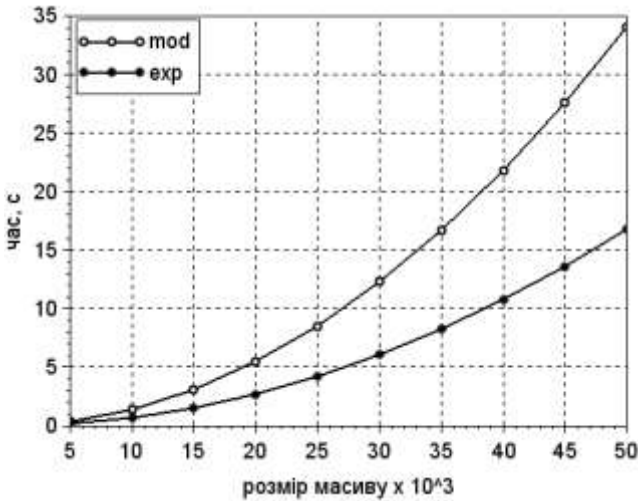


Рисунок 4.1 – Експериментальна і модельна оцінки для послідовного бульбашкового алгоритму сортування

4.2.2 Сортування методом «парний-непарний»

Метод бульбашкового сортування дуже неефективний і на практиці використовується виключно з навчальною метою. Окрім того, цей метод важко розпаралелити.

Розглянемо модифікацію методу бульбашкового сортування, відому як метод перестановки «парний-непарний» (*odd-even transposition*). Метод полягає у тому, що алгоритм сортування ділиться на дві частини. У першій частині (ітерації) порівнюються і, за потреби, міняються місцями елементи масиву з парними і непарними індексами, у другій частині (ітерації) – елементи з непарними і парними індексами. У кожній ітерації виконуються $n/2$ базових операцій порівняння-обміну. Після n ітерацій масив буде гарантовано впорядковано.

Порівняння і обмін пар елементів не залежать одне від одного, тому можуть виконуватися паралельно.

Програма, у якій реалізовано послідовний і паралельний алгоритми сортування "парний-непарний", наведена у лістингу 4.8. За допомогою функції `copy()` масив після сортування послідовним способом переводиться до початкового невпорядкованого виду. Код функцій `fill()` і `check()` запозичено з програми бульбашкового сортування (лістинг 4.7).

Лістинг 4.8 – Програма сортування «парний-непарний»

```

1  #include <iostream>
2  #include <ctime>
3  #include <stdlib.h>
4  #include <omp.h>
5
6  using namespace std;
7
8  void oddevensort(double*, int);
9  void fill(double*, int);
10 void check(double*, int);
11 void copy(double*, double*, int);
12
13 int main()
14 {
15     srand(time(0));
16     int n = 5000;
17     double* d = new double[n];
18     double* a = new double[n];
19     double t;
20
21     printf("array size = %d\n",n);
22     fill(d,n);
23     copy(d,a,n);
24
25     t = omp_get_wtime();
26     oddevensort(a,n);
27     t = omp_get_wtime() - t;
28     check(a,n);
29     printf("serial odd-even sort time = %f sec\n", t);
30     double s = t; // для швидкості
31
32     int num_threads;
```

```

33 int i;
34 int phase;
35 double tmp;
36 copy(d,a,n);
37 #pragma omp parallel
38 {
39     num_threads = omp_get_num_threads();
40 }
41 printf ("num_threads = %d\n", num_threads);
42 t = omp_get_wtime();
43 #pragma omp parallel num_threads(num_threads)
    shared(a,n) private(i, tmp, phase)
44 {
45     for(phase = 0; phase < n; phase++)
46     {
47         if(phase %2 == 0)
48             #pragma omp for
49                 for(i=1; i<n; i+=2)
50                 {
51                     if(a[i-1] > a[i])
52                     {
53                         tmp = a[i-1];
54                         a[i-1] = a[i];
55                         a[i] = tmp;
56                     }
57                 }
58             else
59                 #pragma omp for
60                     for(i=1; i < n-1; i +=2)
61                     {
62                         if(a[i] > a[i+1])
63                         {
64                             tmp = a[i+1];
65                             a[i+1] = a[i];
66                             a[i] = tmp;
67                         }
68                     }
69     }
70 }
71 t = omp_get_wtime() - t;
72 check(a,n);

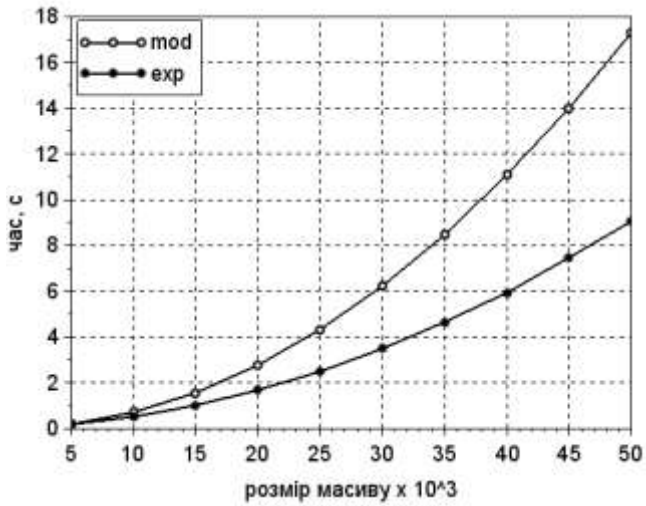
```

```

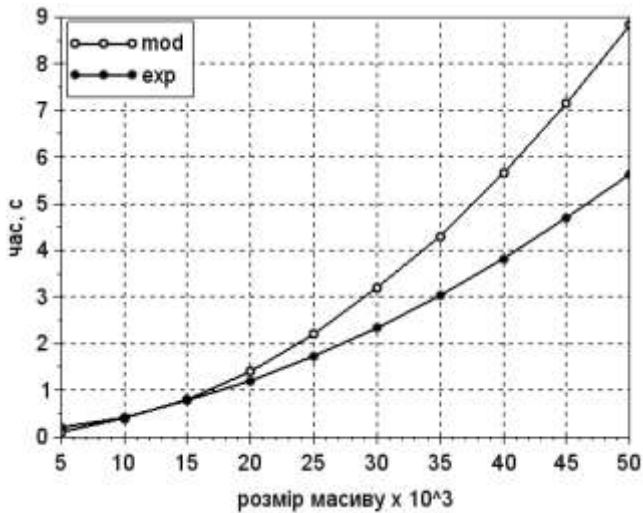
73 printf("parallel odd-even sort time = %f sec,
       speed = %.2f\n", t, s/t);
74
75 delete [] d;
76 delete [] a;
77 return 0;
78 }
79
80 void oddevensort(double* a, int n)
81 {
82     double t;
83     for(int i = 0; i < n; i++)
84     {
85         for(int j = (i%2)? 0 : 1; j < n-1; j+=2)
86         {
87             if(a[j] > a[j+1])
88             {
89                 t = a[j+1];
90                 a[j+1] = a[j];
91                 a[j] = t;
92             }
93         }
94     }
95 }
96
97 void copy(double* d, double* a, int n)
98 {
99     for(int i = 0; i < n; i++)
100         a[i] = d[i];
101 }

```

Порівняння експериментальної і модельної оцінок часових витрат в залежності від розміру сортованого масиву для алгоритму сортування «парний-непарний» для випадків залучення двох і чотирьох потоків на 4-ядерному процесорі AMD A6-3650 2,6 GHz надано на рис. 4.2.



а) два потоки



б) чотири потоки

Рисунок 4.2 – Експериментальна і модельна оцінки для алгоритму сортування «парний-непарний»

4.2.3 Швидке сортування

Алгоритм швидкого сортування, який розробив у 1960 р. англійський математик Ч. Ентоні Р. Хоар, використовується на практиці найчастіше, завдяки високій швидкості сортування, яка для масиву з n елементів у середньому пропорційна $n \cdot \log n$.

Без урахування часових витрат на вибірку елементів масиву з пам'яті до кешу, будемо оцінювати середній час роботи алгоритму за наступною формулою:

$$T = 1.4 \cdot \tau \cdot n \cdot \log_2 n, \quad (4.7)$$

де τ – час виконання однієї операції порівняння-обміну.

Слід, проте, пам'ятати, що мають місце і «найгірші випадки», наприклад, такий розподіл елементів у масиві, що для його сортування необхідно виконати максимальне число операцій порівняння-обміну. Тоді час сортування може виявитися пропорційним $\Theta(n^2)$ [6].

Алгоритм функціонує за принципом «розподіляй і володарюй». На першій ітерації сортований масив ділиться на дві частини і сортування у цих частинах виконуються незалежно одна від одної. Для розділу масиву на частини випадково чи за деякими правилами обирається ведучий (роздільний) елемент, і всі елементи, менші за нього, переносяться до першого блоку (частини), а усі інші – до другого. На наступних ітераціях ці правила застосовуються рекурсивно. Після $\log_2 n$ ітерацій вихідний масив стає впорядкованим.

У наведеній реалізації алгоритму швидкого сортування (див. лістинг 4.9) для наочності коду операцію «порівняння-обмін» оформлено у вигляді двох функцій – `exch()` і `comprexch()` (рядки 59–67). У програмі використовується декілька відомих спеціальних прийомів, що покращують надійність і продуктивність алгоритму:

– у якості роздільного елемента обирається медіана з першого, середнього і останнього елементів (рядки 90–93), що зменшує вірогідність максимально незбалансованого розділу, при якому час сортування може стати пропорційним $\Theta(n^2)$;

– рекурсивний виклик функції сортування відсікається при невеликих розмірах масиву ($m2 \leq 10$, рядок 89) і сортування здійснюється методом вставки (функція `insertion()`, рядки 98–115);

– при розмірі сортованого масиву менше за 1000 елементів сортування виконується в одному потоці (рядки 127–128), що також підвищує загальну швидкодію алгоритму.

Лістинг 4.9 – Програма швидкого сортування

```

1  #include <iostream>
2  #include <stdio.h>
3  #include <time.h>
4  #include <stdlib.h>
5  #include <omp.h>
6
7  using namespace std;
8
9  void fill(double*, int);
10 void copy(double*, double*, int);
11 void check(double*, int);
12 void hybridsort(double*, int, int);
13 void hybridsort_omp_tasks(double*, int, int);
14 static const int m1 = 1000;
15 static const int m2 = 10;
16
17 int main()
18 {
19     srand(time(0));
20     int n = 1000000;
21     double t;
22     double* d = new double[n];
23     double* a = new double[n];
24
25     fill(d,n);
26     copy(d,a,n);
27
28     t = omp_get_wtime();
29     hybridsort(a,0,n-1);
30     t = omp_get_wtime() - t;
31     check(a,n);
32     printf("array size = %d\n",n);

```

```

33  printf("serial quick sort time = %f sec\n",t);
34  double s = t; // для швидкості
35
36  int num_threads;
37  #pragma omp parallel
38  {
39      num_threads = omp_get_num_threads();
40  }
41  printf("num_threads = %d\n", num_threads);
42  copy(d,a,n);
43  t = omp_get_wtime();
44  #pragma omp parallel num_threads(num_threads)
45  {
46      #pragma omp single nowait
47          hybridsort_omp_tasks(a,0,n-1);
48  }
49  t = omp_get_wtime() - t;
50  check(a,n);
51  s = s/t;
52  printf("parallel quick sort time = %f sec ,
        speed = %.2f\n",t,s);
53
54  delete[] d;
55  delete[] a;
56  return 0;
57 }
58
59 void exch(double &a, double &b)
60 {
61     double t = a; a = b; b = t;
62 }
63
64 void compexch(double &a, double &b)
65 {
66     if(b < a) exch(a,b);
67 }
68
69 int partition(double* a, int lo, int hi)
70 {
71     double t;
72     double p = a[hi];
73     int i = lo-1;

```

```

74     int j = hi;
75     for(;;)
76     {
77         while(a[++i] < p);
78         while(p < a[--j])
79             if(j == lo) break;
80             if(i >= j) break;
81         exch (a[i],a[j]);
82     }
83     swap (a[i],a[hi]);
84     return i;
85 }
86
87 void quicksort(double* a, int lo, int hi)
88 {
89     if(hi -lo <= m2) return;
90     exch(a[(lo+hi)/2],a[hi - 1]);
91     compexch(a[lo],a[hi - 1]);
92     compexch(a[lo],a[hi]);
93     compexch(a[hi - 1], a[hi]);
94     int i = partition(a, lo+1, hi-1);
95     quicksort(a,lo,i-1);
96     quicksort(a,i+1,hi);
97 }
98 void insertion(double* a, int lo, int hi)
99 {
100     int i;
101     double t;
102     for(i = hi; i > lo; i--)
103         compexch(a[i-1],a[i]);
104     for(i = lo+2; i<= hi; i++)
105     {
106         int j = i;
107         t = a[i];
108         while(t < a[j-1])
109         {
110             a[j] = a[j-1];
111             j--;
112         }
113         a[j] = t;
114     }
115 }

```

```

116
117 void hybridsort(double* a, int lo, int hi)
118 {
119     quicksort(a, lo, hi);
120     insertion(a, lo, hi);
121 }
122
123 void hybridsort_omp_tasks(double *a, int lo, int hi)
124 {
125     if(lo < hi)
126     {
127         if(hi - lo < m1)
128             hybridsort(a, lo, hi);
129         else
130         {
131             int i = partition(a, lo, hi);
132             #pragma omp task
133             hybridsort_omp_tasks(a, lo, i - 1);
134             hybridsort_omp_tasks(a, i + 1, hi);
135         }
136     }
137 }

```

Експериментальну і модельну оцінки залежності часу для алгоритму швидкого сортування від розміру масиву на 4-ядерному процесорі AMD A6-3650 2,6 GHz надано на рис. 4.3.

4.3 Завдання до лабораторної роботи

4.3.1 Завдання до розділу 4.1.

4.3.1.1 Реалізувати та вивчити програму, що наведена у лістингу 4.2. Визначити версію бібліотеки OpenMP, встановленої на комп'ютері.

4.3.1.2 Реалізувати та вивчити програму, що наведена у лістингу 4.3. Модифікувати програму таким чином, щоб вона циклічно виконувалася для усіх непарних $i < 10$.

4.3.1.3 Реалізувати та вивчити програму, що наведена у лістингу 4.4. Продемонструвати працездатність програми для числа потоків 3 і 5.

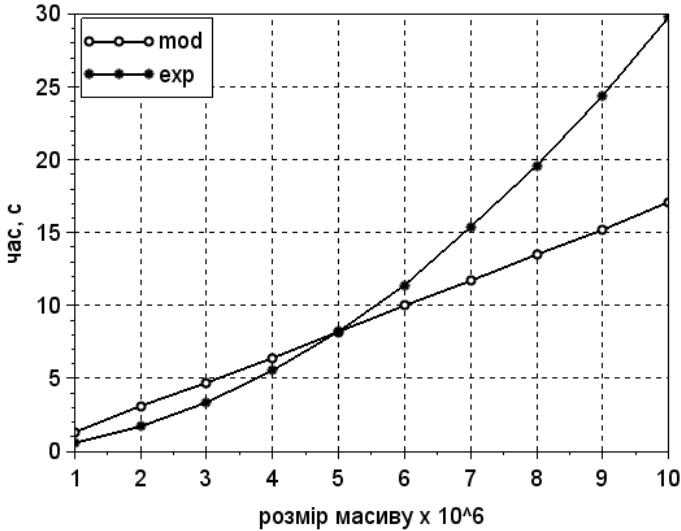


Рисунок 4.3 – Експериментальна і модельна оцінки для алгоритму швидкого сортування

4.3.1.4 Модифікувати програму, що наведена у лістингу 3.5, наступним чином:

- матриці повинні мати тип `double` та розміщуватися у динамічній пам'яті;
- створити функцію заповнення елементів масиву випадковими числами (див. лістинг 4.7);
- у програмі реалізувати можливість виміру часу обчислення добутку матриць. Для цього використати функцію `omp_get_wtime()` (див. лістинг 4.6);
- провести експерименти з модифікованою програмою та побудувати графіки залежності часу послідовного і паралельного обчислень добутку матриць від розміру матриць, змінюючи їх розмір з кроком 250 у діапазоні $[250, 2000]$;
- побудувати графік залежності прискорення паралельних обчислень від розміру матриць.

4.3.1.5 Реалізувати програму, що наведена у лістингу 4.6. Побудувати графіки залежності часу послідовного і паралельного обчислень, а також прискорення від числа ітерацій. Крок зміни числа ітерацій – $0.25 \cdot 10^6$, а діапазон зміни – $[0.25 \cdot 10^6, 2 \cdot 10^6]$.

4.3.1.6 Розробити програму, яка обчислює значення π згідно наступної формули апроксимації:

$$\pi = 4 \cdot \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right] = 4 \cdot \sum_{i=0}^{\infty} \frac{(-1)^i}{2 \cdot i + 1}. \quad (4.8)$$

Програма має використовувати директиву `for` з опцією редукації. Здійснити виміри характеристик реалізації і побудувати графіки аналогічно до завдання 4.3.1.5.

4.3.2 Завдання до розділу 4.2.

4.3.2.1 Реалізувати і вивчити програму, що наведена у лістингу 4.7. Провести наступні дослідження:

- експериментально оцінити час, необхідний для виконання однієї операції «порівняти-обмінати» на Вашій робочій станції;

- виконати необхідні обчислення і обчислювальні експерименти, побудувати графіки, аналогічні до наведених на рис. 4.1. Для модельної оцінки часу послідовного сортування використати наступні фізичні характеристики комп'ютера: латентність $\alpha \approx 93,0$ нс, ширина шини пам'яті $\omega = 64$, її пропускна спроможність $\beta \approx 6400$ Мб/с.

4.3.2.2 Реалізувати і вивчити програму сортування «парний-непарний», що наведена у лістингу 4.8. Провести наступні дослідження:

- виконати необхідні обчислення і обчислювальні експерименти, побудувати графіки, аналогічні до наведених на рис. 4.2, для послідовного сортування і паралельного на двох потоках;

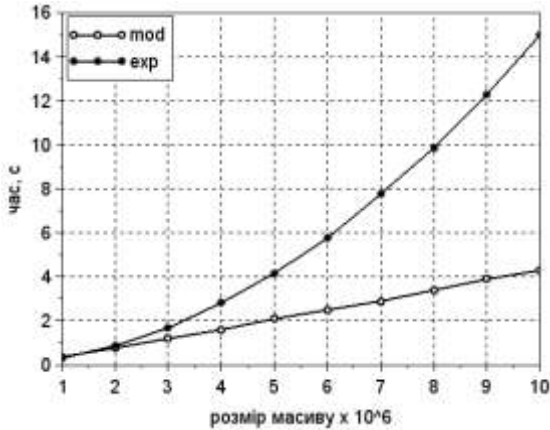
- побудувати графік залежності прискорення паралельного сортування від розміру сортованого масиву.

4.3.2.3 Реалізувати і вивчити програму швидкого сортування, наведену у лістингу 3.9. Провести наступні дослідження:

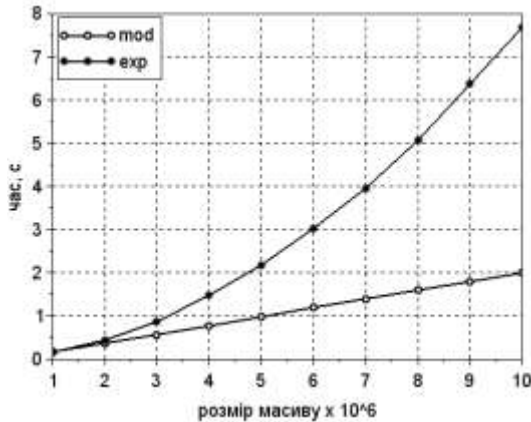
- виконати необхідні обчислення і обчислювальні експерименти, побудувати графіки, аналогічні до наведених на

рис. 4.4, для модельної і експериментальної оцінок часових витрат на паралельне швидке сортування на двох потоках;

– побудувати графік залежності прискорення паралельного сортування від розміру сортованого масиву.



а) два потоки



б) чотири потоки

Рисунок 4.4 – Експериментальна і модельна оцінки залежності часу паралельного швидкого сортування від розміру масиву на 4-ядерному процесорі AMD A6-3650 2,6 GHz

4.4 Зміст звіту

- 4.4.1 Титульний лист.
- 4.4.2 Мета роботи.
- 4.4.3 Фрагменти програмного коду, розроблені самостійно.
- 4.4.4 Графіки, побудовані згідно завдань.
- 4.4.5 Відповіді на контрольні питання. Вміти їх пояснювати в усній формі.

4.5 Контрольні питання

- 4.5.1 Призначення бібліотеки OpenMP.
- 4.5.2 Чим відрізняються організації паралельних обчислень за допомогою MPI і OpenMP?
- 4.5.3 Що таке директива в OpenMP і які види директив визначено в OpenMP?
- 4.5.4 Що таке структурний блок в OpenMP і яке його призначення?
- 4.5.5 Скільки потоків створюється у паралельній області OpenMP-програми за замовчанням?
- 4.5.6 Яка версія OpenMP використовується на Вашій робочій станції, як її визначити?
- 4.5.7 Пояснити модель програми у середовищі OpenMP.
- 4.5.8 Яка модель даних визначена в OpenMP?
- 4.5.9 Якими способами можна розподілити обчислення між потоками?
- 4.5.10 Якими способами і за допомогою яких опцій можна розподілити обчислення за ітераціями, використовуючи директиву `for`?
- 4.5.11 Для чого і як використовується опція редукції `reduction`?
- 4.5.12 Для чого використовується директива `section`?
- 4.5.13 Призначення директиви `task`.
- 4.5.14 Призначення директив `single` і `master`.
- 4.5.15 Перелічити та пояснити призначення директив синхронізації.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Уильямс, Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ / Э. Уильямс. – ДМК Пресс, 2012.
2. Хорстман, К. С. Java. Библиотека профессионала, том 1. Основы. 9-е изд. / К. С. Хорстман, Г. Корнелл. – М: Вильямс, 2014.
3. Антонов, А. С. Параллельное программирование с использованием технологии MPI: учеб. пособие / А. С. Антонов. – М: МГУ, 2004.
4. Антонов, А. С. Параллельное программирование с использованием технологии OpenMP: учеб. пособие / А. С. Антонов. – М: МГУ, 2009.
5. Гергель, В. П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем: учеб. пособие / В. П. Гергель. – Нижний Новгород: ННГУ, 2010.
6. Седжвик, Р. Фундаментальные алгоритмы на C++ / Р. Седжвик. – СПб: ДиаСофтЮП, 2002.
7. Левин, М. П. Параллельное программирование с использованием OpenMP: учеб. пособие / М. П. Левин. – М: БИНОМ, 2012.
8. Параллельные вычисления CUDA. URL: <http://www.nvidia.com.ua/object/cuda-parallel-computing-ru.html>;
9. Distributed Computing team of Ukraine – сайт Украинской команды распределенных вычислений. URL: <http://distributed.org.ua>.
10. Open MPI: Open Source High Performance Computing. URL: <https://www.open-mpi.org/>. (Last accessed: 02.01.2020)
11. MPI Forum. URL: <https://www.mpi-forum.org/>. (Last accessed: 02.01.2020)
12. The OpenMP API specification for parallel programming. URL: <https://www.openmp.org/>. (Last accessed: 02.01.2020)
13. Stéphane Ethier. Introduction parallel programming using MPI and OpenMP. *Presentation slides*. URL: https://w3.pppl.gov/~ethier/PICSCIE/MPI_OpenMP_tutorial_Dec_2017.pdf (Last accessed: 02.01.2020)

ДОДАТОК А.

Порядок створення MPI-додатку у середовищі DEV-C++

А.1 Завантажте Dev-C++ (**Start Menu|All Programs|Bloodshed Dev-C++|Dev-C++**).

А.2 Створіть новий проект (**File|New|Project...**). Вкажіть тип програми **Console Application**, тип проекту **C++ Project**, задайте ім'я проекту у полі **Name**, натисніть **OK** та вкажіть шлях до робочої папки, що буде містити проект.

А.3 Включіть до проекту бібліотеку MPI, для цього в головному меню Dev-C++ оберіть пункт **Project|Project Options...** У вікні, що відкриється, оберіть підрозділ **Parameters**, натисніть «**Add Library or Object**» та вкажіть шлях до **MPICH2\lib\mpi.lib**. Далі оберіть підрозділ **Directories ...**, пункт **Include Directories** та вкажіть шлях до папки **MPICH2\include**. Натисніть кнопку **Add**.

А.4 У дереві проекту в лівій частині вікна Dev-C++ оберіть модуль **main.cpp** та напишіть код програми.

А.5 Для компіляції проекту натисніть **F12** або оберіть пункт **Execute|Rebuild All** головного меню програми.

ДОДАТОК Б.

Порядок запуску MPI-додатку за допомогою WMPIEXEC

Б.1 Скопіюйте ваш MPI-додаток у папку MPI_shared. Це забезпечує видимість файлів з усіх вузлів комп'ютерної мережі.

Б.2 Завантажте wmpiexec (**Start Menu|All Programs|MPICH2|wmpiexec.exe**).

Б.3 У полі **Application** вкажіть мережевий шлях до **exe**-файлу вашого додатку, наприклад, \\ws23\MPI_shared\lab4_1.exe

Б.4 Оберіть опцію **more options** для відображення додаткових налаштувань.

Б.5 Для локального запуску MPI-додатку на ядрах процесора вашого хоста у полі **extra mpiexec options** вкажіть ключ **-localonly**

Б.6 Для запуску MPI-додатку на процесорах комп'ютерів мережі у полі **hosts** через прогалину вкажіть доменні імена комп'ютерів мережі, на яких буде виконуватися MPI-додаток (наприклад, **ws23 ws25**). При цьому ключа **-localonly** не повинно бути.

Б.7 У полі **Number of processes** вкажіть кількість процесів, на яких буде виконуватися MPI-додаток (1, 2, 3, 4).

Б.8 Для запуску додатку натисніть кнопку **Execute**. При цьому виведення **stdout** на консоль та введення **stdin** будуть виконуватися у вікні wmpiexec. **Примітка:** для вводу даних у додаток користуйтеся комбінацією клавіш **Ctrl+Enter** замість **Enter**. Інколи після натискання кнопки **Execute** також потрібно натиснути **Ctrl+Enter**. Це пов'язано з буферизацією потоків **stdout** та **stdin**.

Б.9 Для перевірки прав користувача на віддалений запуск додатку необхідно у полі, що з'явиться перед запуском, ввести ім'я користувача у форматі домен\користувач (наприклад, fknt353.zp.edu.ua\fknt517) та пароль входу до системи. Можна натиснути кнопку **«Register»** щоб зареєструвати права у реєстрі операційної системи. Діалог реєстрації прав користувача можна викликати за допомогою утиліти **Start Menu|All Programs|MPICH2|wmpiregister.exe**

Б.10 Для відлагодження додатку користуйтеся локальним запуском на одному та двох процесорах. Після цього запускайте додаток згідно завдання на одному, двох та чотирьох процесорах. Як правило використовується локальний хост та сусідні хости.