

Приклад MPI програми на C

1. Опис

Дана програма призначена для розпаралелювання задачі знаходження суми всіх елементів деякої матриці між декількома процесами.

Основна ідея полягає у рівномірному розділенні даної матриці на частини (по рядкам) поміж процесами, які кожний процес може обробляти самостійно (в нашому випадку підраховувати суму всіх елементів у відповідній своїй частині матриці) і частковий результат виконання відправляти на сторону root (головного) – процесу, який вже визначає остаточний результат роботи програми (суму всіх часткових сум).

Програма написана у відповідності до стандарту MPI для C.

2. Лістинг програми Example_MPI_C.cpp:

```
1. #include <mpi.h>
2.
3. #include <iostream>
4. #include <iomanip>
5.
6. #include <cstdlib>
7.
8. #include <vector>
9. #include <numeric>
10. #include <algorithm>
11.
12.
13. template< typename T >
14. class dynamic_matrix // клас динамічної матриці
15. {
16.     public:
17.         typedef typename std::vector<T>::size_type size_type;
18.
19.     public:
20.         dynamic_matrix(size_type m, size_type n): matr(m * n), n(n) {} // створюємо матрицю m x n
21.
22.         dynamic_matrix(): n(0) {} // створюємо матрицю 0 x 0
23.
24.         void resize(size_type m, size_type n) { matr.resize(m * n); this->n = n; } // змінюємо розмір на m x n
25.
26.         T * begin() { return &matr.front(); } // повертаємо вказівник на початок,
27.         T * end() { return &matr.back() + 1; } // кінець матриці
28.         T * operator[](size_type i) { return &matr[i * n]; } // повертаємо вказівник на i - тий рядок матриці
29.
30.     private:
31.         std::vector<T> matr;
32.         size_type n;
33. };
34.
35.
36. int main(int argc, char **argv)
37. {
38.     int errorkod = MPI_Init(&argc, &argv); // ініціалізуємо MPI
39.
40.     if(errorkod)
41.     {
42.         std::cerr << "\nError starting MPI programm!\n";
43.         MPI_Abort(MPI_COMM_WORLD, errorkod); // завершуємо роботу всіх процесів у разі невдачі ініціалізації
44.     }
45.
46.     int NumberProcesses; // число запущених процесів користувачем
47.     int id; // номер процесу
48.
49.     MPI_Comm_size(MPI_COMM_WORLD, &NumberProcesses); // знаходимо число запущених процесів
50.     MPI_Comm_rank(MPI_COMM_WORLD, &id); // визначення процесом свого номера
51.
52.     int n; // порядок матриці
53.     double t1, t2;
54.
55.     if(!id) // root - процес id == 0
56.     {
57.         while(1)
58.         {
59.             std::cout << "Please, enter the order of the matrix: " << std::flush;
60.             std::cin >> n; // вводим порядок матриці
61.
62.             if(n > 0 && std::cin)
63.                 break;
64.
65.             std::cout << "\nError: Order of the matrix must > 0!\n";
66.
67.             std::cin.clear();
68.             std::cin.sync();
69.         }

```

```

70.
71.     t1 = MPI_Wtime(); // фіксуємо початковий момент виконання
72. }
73.
74. MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); // розсилаємо значення порядку матриці (n) всім процесам
75.
76.
77. int NumberActiveProcesses = std::min(NumberProcesses, n); // визначаємо число діючих процесів
78.
79. int m = n / NumberActiveProcesses + (id >= NumberActiveProcesses - n % NumberActiveProcesses); // знаходимо
80. // кількість рядків матриці, що припадають на даний процес (із номером id)
81.
82. long long sum = 0;
83.
84. dynamic_matrix<int> matr;
85.
86. std::vector<MPI_Request> array_of_requests; // масив комунікаційних об'єктів для неблокуючого обміну, потрібний для
87. // функції MPI_Isend (далі), більше не використовується
88.
89. if(!id) // root - процес
90. {
91.     matr.resize(n, n); // утворюємо матрицю n x n з
92.     std::generate(matr.begin(), matr.end(), rand); // випадковими значеннями елементів
93.
94.     array_of_requests.resize(NumberActiveProcesses - 1);
95.
96.     int i = 1;
97.     int row = 0;
98.     int m_slv;
99.
100.    for(; i < NumberActiveProcesses; ++i)
101.    {
102.        m_slv = n / NumberActiveProcesses + (i >= NumberActiveProcesses - n % NumberActiveProcesses); // знаходимо
103.        // кількість рядків матриці i - го slave (підрядного) - процесу
104.
105.        MPI_Isend(matr[row], m_slv * n, MPI_INT, i, i, MPI_COMM_WORLD, &array_of_requests[i - 1]); // посилаємо
106.        // у фоновому режимі i - му slave - процесу частину матриці m_slv x n
107.
108.        row += m_slv;
109.    }
110.
111.    sum = std::accumulate(matr[row], matr.end(), sum); // знаходимо суму елементів частини матриці, що припадає на
112.    //root - процес
113. }
114. else // slaves - процеси id != 0
115. if(id < NumberActiveProcesses) // для діючих slaves - процесів
116. {
117.     matr.resize(m, n); // створюємо відповідну матрицю - буфер
118.
119.     MPI_Recv(matr[0], m * n, MPI_INT, 0, id, MPI_COMM_WORLD, &MPI_Status()); // приймаємо частину матриці, що
120.     // припадає на даний slave - процес
121.
122.     sum = std::accumulate(matr.begin(), matr.end(), sum); // знаходимо суму елементів відповідної частини матриці
123. }
124.
125.
126. long long total_sum;
127.
128. MPI_Reduce(&sum, &total_sum, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD); // посилаємо часткові результати (sum)
129. // з усіх процесів до root - процесу, де вони зразу ж сумуються і записуються у змінну total_sum
130.
131.
132. if(!id)
133. {
134.     t2 = MPI_Wtime(); // фіксуємо час кінця виконання
135.
136.     int i = 0;
137.     int j;
138.
139.     std::cout << "\nMatrix:\n\n";
140.
141.     for(; i < n; ++i)
142.     {
143.         for(j = 0; j < n; ++j)
144.             std::cout << std::setw(11) << matr[i][j];
145.
146.         std::cout << '\n';
147.     }
148.
149.     std::cout << "\nTotal sum of all elemets is " << total_sum;
150.     std::cout << "\nCalculation time is " << (t2 - t1) << " seconds\n";
151. }
152.
153.
154. MPI_Finalize(); // завершуємо роботу в MPI
155.
156. return 0;
157. }
158.

```

3. Аналіз

З 1. – 10. рядок коду програми підключаються відповідні заголовочні файли, зокрема для користування бібліотекою MPI підключені в 1 – му рядку файл `mpi.h`.

З 13. – 33. рядок об'являється клас `dynamic_matrix`, який дозволяє динамічно задавати і змінювати розмірність матриці.

У 38. рядку ініціалізується середовище MPI шляхом звернення до функції `MPI_Init`, звернення до якої має бути першим і єдиним у MPI програмі.

З 40. – 44. рядок перевіряємо як пройшла ініціалізація, якщо спіткала невдача з тої чи іншої причини, завершуємо роботу всіх процесів за допомогою функції `MPI_Abort`, яку передаємо в якості 1 – го аргументу глобальний комунікатор `MPI_COMM_WORLD` (ідентифікує групу запущених процесів і область обміну між ними), а в якості 2 – го аргументу код помилки.

У рядку 49. і 50. кожний процес знаходить розмір (`NumberProcesses`) і свій номер `id` є `[0, NumberProcesses - 1]` у групі, в якій він запущений, відповідно, функції `MPI_Comm_size` і `MPI_Comm_rank`.

При чому на далі вважатимемо, що `id == 0` має `root` (головний) – процес, а всі інші процеси із `id != 0` є `slaves` (підрядні) – процеси.

З 57. – 69. рядок вводиться і перевіряється на коректність, у `root` – процесі, порядок матриці.

У 71. рядку фіксуємо початковий момент часу, за допомогою функції `MPI_Wtime`, з якого і починається розпаралельовання.

У 74. рядку розсилаємо значення порядку матриці від `root` – процесу всім процесам за допомогою функції `MPI_Bcast`, в якій перші 3 параметри характеризують відповідно адрес, кількість і тип даних; 4 – ий параметр показує номер `root` – процесу, а 5 – ий – комунікатор групи процесів.

У 77. рядку визначаємо число діючих процесів, у випадку коли користувач запустив програму із числом процесів більшим, а ніж введений порядок матриці, то використовуються не всі процеси – їх кількість дорівнює порядку матриці. Тобто на кожний процес буде приходиться по одному рядку матриці.

У 79. рядку кожний процес визначає кількість рядків матриці, що припадають на нього. У випадку не кратності числа процесів порядку матриці, рядки матриці рівномірно перерозподіляються між процесами, починаючи із останнього.

З 89. – 113. рядок `root` – процес створює матрицю відповідного порядку із випадковими числами, частини якої розсилає активним `slaves` – процесам, у відповідності до розподілення числа рядків між ними. При чому розсилка відбувається паралельно (функція `MPI_Isend`) з підрахуванням суми елементів своєї частини матриці у змінну `sum`. Функція `MPI_Isend` має перші 3 параметри відповідно адрес, кількість і тип даних, що посилаються; 4 – ий параметр показує номер процесу – адресата, 5 – ий – ідентифікатор повідомлення, 6 – ий – комунікатор групи процесів, а 7 – ий – адрес комунікаційного об'єкта.

З 115. – 124. рядок кожний активний `slave` – процес створює матрицю – буфер для прийому відповідної частини головної матриці, що припадає на нього, після чого сумує всі елементи у змінну `sum`. Для прийому використовується функція `MPI_Recv`, в якій перші 3 параметри відповідно адрес, кількість і тип даних, що приймаються; 4 – ий параметр показує номер процесу – адресанта, 5 – ий – ідентифікатор повідомлення, 6 – ий – комунікатор групи процесів, а 7 – ий – адрес об'єкта атрибутів повідомлення.

У 128. рядку кожний процес посилає частковий результат підрахованої суми (змінна `sum`) до `root` – процесу, де вони остаточно сумуються і записуються у змінну `total_sum`. Дана операція реалізовується за допомогою функції `MPI_Reduce`, яка в якості 1 – го параметра приймає адрес буфера посилки, в якості 2 – го – адрес буфера результату у `root` – процесі, 3 і 4 – ий параметр відповідно кількість і тип даних у буфері посилки, 5 – ий – операція редукції, яка буде виконуватися на стороні `root` – процесу (в нашому випадку `MPI_SUM`), 6 – ий – номер `root` – процесу і 7 – ий – комунікатор групи процесів.

У 134. рядку фіксуємо кінцевий момент часу, за допомогою функції `MPI_Wtime`.

З 136. – 151. рядок друкуємо головну матрицю і остаточні результати виконання програми.

У 154. рядку завершуємо роботу в MPI шляхом звернення до функції `MPI_Finalize`, звернення до якої має бути останнім і єдиним у MPI програмі.

4. Компіляція і компоновка

Для компіляції коду даної програми потрібно прописати у терміналі:

```
mpicc -c Example_MPI_C.cpp
```

Для компоновки у виконавчий файл `Example_MPI_C`:

```
mpicc -o Example_MPI_C Example_MPI_C.o
```

5. Запуск

Для запуску програми `Example_MPI_C`, наприклад, на 4 – х вузлах кластера із використанням менеджера ресурсів `slurm`, треба прописати у терміналі:

```
srun -N 4 -A
```

внаслідок чого виділяється в розпорядження 4 вузла кластера і запускається допоміжний shell, в якому і запускаємо MPI програму:

```
mpirun -np 4 Example_MPI_C
```

для виходу із допоміжного shell у терміналі прописуємо:

```
exit
```

Автор: Попов Олександр НТУУ "КПІ", Центр Суперкомп'ютерних Обчислень

<mailto:nevrazlyvy@gmail.com>