Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №6:**

**«Коллективные операции в MPI»**

Антон Гатченко Б22-525

2024 г.

*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
* IDE – GCC/G++ 13.1, OpenMP 201511, Microsoft MPI Version 10.1.12498.16

*Ход работы:*

В ходе работы проводились замеры времени выполнения программы, написанной с использованием технологии MSMPI по сортировке массива целочисленных чисел (int 4 byte) размером элементов. Эти результаты сравнивались с результатами программы с OpenMP.

Обе программы были скомпилированы с флагом Ofast.

Так как в сортировке Шелла следующий результат прохода зависит от предыдущего, был использован tile-based принцип для распараллеливания. Исходный массив был поделен на блоки, которые сортировались с помощью сортировки Шелла. Затем блоки объединялись с помощью функции слияния.

Поскольку MPI затрачивает довольно много времени на пересылку массивов, было выполнено таймирование конкретных частей, где происходила сама сортировка, чтобы сравнить плюсы от распараллеливания без влияния этой особенности реализации MPI. Также следует обратить внимание на то, что по каким-то причинам MSMPI сразу завершал выполнение программы, если хоть где-то инициализировался std::vector или использовалась функция std::inplace\_merge. Из-за этого была использована самописная функция inplace\_merge, а также все операции выполнялись на массивах int\*, а не на векторах.

*Использованные функции MSMPI:*

* MPI\_Init() – инициализирует среду MPI и должна быть первой функцией, вызываемой в любом MPI-приложении.
* MPI\_Comm\_size() – возвращает общее количество процессов в данном коммуникаторе (группе созданных MPI\_Init процессов), который чаще всего является глобальным коммуникатором MPI\_COMM\_WORLD.
* MPI\_Comm\_rank() – возвращает уникальный идентификатор процесса из заданного коммуникатора.
* MPI\_Reduce() – собирает данные от всех процессов и применяет к ним указанную операцию (максимум в данной программе), после чего результат передается указанному процессу.
* MPI\_Barrier() – каждый процесс останавливается на этой линии, пока все остальные процессы не дойдут до неё.
* MPI\_Wtime() – возвращает текущее время в секундах, используется для измерения времени выполнения программы.
* MPI\_Finalize() – завершает работу MPI и освобождает ресурсы, связанные с MPI. Должна быть последней вызванной функцией в MPI-программе.

*Выбор функций для обмена данными:*

Для распределения и сбора частей массива в первой итерации (в функции tile\_based\_shell\_sort) были использованы следующие функции:

* MPI\_Scatter() – автоматически отправляет равные части массива каждому процессу в зависимости от rank, что упрощает коммуникации для нашего случая, поскольку данные равномерно распределены.
* MPI\_Gather() – используется для сбора всех отсортированных частей массива на заданный процесс.

В функции parallel\_merge\_blocks использовались:

* MPI\_Send() – для точечной рассылки от потока к потоку. Так как количество потоков, объединяющих блоки, постоянно уменьшалось, нужно было более гибкое управление, чем Scatter.
* MPI\_Recv() – для получения информации от MPI\_Send.

*Графики выполнения программы при различном количестве потоков и сравнение с OpenMP-версией:*

*Заключение:*

В ходе данной лабораторной работы был изучен процесс использования технологии MPI для организации параллельных вычислений на примере функции сортировки Шелла. Программа была запущена при разном количестве потоков, что позволило оценить влияние распараллеливания на производительность, а время выполнения и иные показатели сравнивались с OpenMP-версией программы.

Результаты тестов показали, что MPI оказалась более эффективной в данной задаче, хотя из-за особенностей таймирования были исключены некоторые операции помимо распределения памяти между потоками, что могло бы сказаться на результатах (но все основные операции для сортировки были учтены). На 2 и 3 потоках ускорение было сверхлинейным, больше, чем число потоков, но ненамного.

Следует отметить, что само время выполнения в варианте на MPI оказалось немного ниже, чем на OpenMP. Без оптимизирующих флагов время на одном потоке для варианта с OpenMP было чуть ниже 12 секунд, для MPI - ниже 7 секунд (ускорение и эффективность были примерно теми же, что и с флагом).

Замедление роста производительности у обоих вариантах при увеличении числа потоков также может быть вызвано сторонними процессами системы и влиянием планировщика ОС на выделяемые программе ресурсы.

*Приложение:*

1. Исходный код программы с измерением времени работы программы:

#include <cstdlib>  
#include <cstdio>  
#include <mpi.h>  
#include <algorithm>  
#include <iostream>  
#include <random>  
#include <vector>  
  
#define SEED 920215  
#define SEED\_INC 12345  
#define RUNS\_NUM 10  
#define SIZE static\_cast<int>(1e7)  
  
using std::swap, std::min;  
  
void generate\_random\_array(int \*array, const int seed){  
 std::mt19937\_64 gen(seed);  
 std::uniform\_int\_distribution dist(INT\_MIN, INT\_MAX);  
  
 for (int i = 0; i < SIZE; i++){  
 array[i] = dist(gen);  
 }  
}  
  
void shell\_sort(int \*array, const int size){  
 for (int gap = size / 2; gap > 0; gap /= 2){  
 for (int i = gap; i < size; ++i){  
 for (int j = i - gap; j >= 0 && array[j] > array[j + gap]; j -= gap){  
 swap(array[j], array[j + gap]);  
 }  
 }  
 }  
}  
  
void inplace\_merge(int \*array, int size){  
 int mid = size / 2;  
 int left\_size = mid + 1;  
 int right\_size = size - mid;  
  
 // Создаем временные массивы для левого и правого подмассивов  
 int \*left\_array = (int \*) malloc(left\_size \* sizeof(int));  
 int \*right\_array = (int \*) malloc(right\_size \* sizeof(int));  
  
 // Копируем данные во временные массивы  
 memcpy(left\_array, &array[0], left\_size \* sizeof(int));  
 memcpy(right\_array, &array[mid + 1], right\_size \* sizeof(int));  
  
 // Сливаем два отсортированных временных массива обратно в исходный  
 int i = 0, j = 0, k = 0;  
 while (i < left\_size && j < right\_size){  
 if (left\_array[i] <= right\_array[j]){  
 array[k++] = left\_array[i++];  
 } else {  
 array[k++] = right\_array[j++];  
 }  
 }  
  
 // Копируем оставшиеся элементы из левого подмассива (если есть)  
 while (i < left\_size){  
 array[k++] = left\_array[i++];  
 }  
  
 // Копируем оставшиеся элементы из правого подмассива (если есть)  
 while (j < right\_size){  
 array[k++] = right\_array[j++];  
 }  
  
 // Освобождаем временные массивы  
 free(left\_array);  
 free(right\_array);  
}  
  
double parallel\_merge\_blocks(int \*array, int proc\_num){  
 int rank = -1;  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 int block\_size;  
 double time = 0;  
 double start = 0;  
 double end = 0;  
  
 while (proc\_num > 1){  
 proc\_num /= 2;  
 block\_size = SIZE / proc\_num;  
  
 if (rank >= proc\_num){  
 break;  
 }  
  
 int \*local\_array = static\_cast<int \*>(malloc(block\_size \* sizeof(int)));  
  
 if (!rank){  
 for (int i = 1; i < proc\_num; ++i){  
 MPI\_Send(array + i \* block\_size, block\_size, MPI\_INT, i, 0, MPI\_COMM\_WORLD); // S1  
 }  
 start = MPI\_Wtime();  
// std::sort(array, array + block\_size);  
 inplace\_merge(array, block\_size);  
 end = MPI\_Wtime();  
 } else {  
 MPI\_Recv(local\_array, block\_size, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); // R1  
 start = MPI\_Wtime();  
// std::sort(local\_array, local\_array + block\_size);  
 inplace\_merge(local\_array, block\_size);  
 end = MPI\_Wtime();  
 }  
  
 time += end - start;  
  
 if (!rank){  
 for (int i = 1; i < proc\_num; ++i){  
 MPI\_Recv(array + i \* block\_size, block\_size, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); // R2  
 }  
 } else {  
 MPI\_Send(local\_array, block\_size, MPI\_INT, 0, 0, MPI\_COMM\_WORLD); // S2  
 }  
 }  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 return time;  
}  
  
double tile\_based\_shell\_sort(int \*array, const int &proc\_num){  
 int rank = -1;  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
  
 const int block\_size = SIZE / proc\_num;  
 int \*local\_array = static\_cast<int \*>(malloc(block\_size \* sizeof(int)));  
  
 MPI\_Scatter(array, block\_size, MPI\_INT, local\_array, block\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);  
  
 const double start1 = MPI\_Wtime();  
 shell\_sort(local\_array, block\_size);  
 const double end1 = MPI\_Wtime();  
  
 MPI\_Gather(local\_array, block\_size, MPI\_INT, array, block\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);  
  
 double merge\_time = 0;  
  
 if (proc\_num > 1){  
 merge\_time = parallel\_merge\_blocks(array, proc\_num);  
 }  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
  
 return end1 - start1 + merge\_time;  
}  
  
void time\_algorithm(){  
 double timeSpent = 0;  
 double maxTimeSpent = 0;  
 int seed = SEED;  
 int proc\_num = -1;  
 int rank = -1;  
 int \*array = nullptr;  
  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);  
  
 if (!rank){  
 array = static\_cast<int \*>(malloc(SIZE \* sizeof(int)));  
 }  
  
 for (int i = 0; i < RUNS\_NUM; i++){  
 if (!rank){  
 generate\_random\_array(array, seed);  
 seed += SEED\_INC;  
 }  
 timeSpent += tile\_based\_shell\_sort(array, proc\_num);  
 }  
  
 MPI\_Reduce(&timeSpent, &maxTimeSpent, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);  
  
 if (!rank){  
 printf("%lf", timeSpent / RUNS\_NUM);  
 }  
}  
  
int main(int argc, char \*\*argv){  
 int ret = -1; ///< For return values  
 int rank = -1; ///< This processor's number  
  
 ret = MPI\_Init(&argc, &argv);  
 if (!rank){ printf("MPI Init returned (%d)\n", ret); }  
  
 time\_algorithm();  
  
 ret = MPI\_Finalize();  
  
 return 0;  
}

1. Таблица времени работы программы

|  |  |  |
| --- | --- | --- |
| Время работы программы | | |
| **Количество потоков** | **OMP** | **MPI** |
| 1 | 1,7486 | 1,509347 |
| 2 | 0,8706 | 0,754272 |
| 3 | 0,6197 | 0,489066 |
| 4 | 0,4977 | 0,42023 |
| 5 | 0,48 | 0,327667 |
| 6 | 0,4379 | 0,300273 |
| 7 | 0,4114 | 0,277527 |
| 8 | 0,3811 | 0,299669 |
| 9 | 0,3843 | 0,265651 |
| 10 | 0,3608 | 0,250669 |
| 11 | 0,3469 | 0,230467 |
| 12 | 0,342 | 0,223341 |

1. Таблица ускорения программы

|  |  |  |
| --- | --- | --- |
| Ускорение программы | | |
| **Количество потоков** | **OMP** | **MPI** |
| 1 | 1 | 1 |
| 2 | 2,008499885 | 2,001064603 |
| 3 | 2,821687914 | 3,086182642 |
| 4 | 3,513361463 | 3,591716441 |
| 5 | 3,642916667 | 4,606344246 |
| 6 | 3,993149121 | 5,026582477 |
| 7 | 4,250364609 | 5,438559131 |
| 8 | 4,588297035 | 5,036713841 |
| 9 | 4,550091075 | 5,681691392 |
| 10 | 4,846452328 | 6,021275068 |
| 11 | 5,040645719 | 6,549080779 |
| 12 | 5,112865497 | 6,758038157 |

1. Таблица эффективности программы

|  |  |  |
| --- | --- | --- |
| Эффективность программы | | |
| **Количество потоков** | **OMP** | **MPI** |
| 1 | 1 | 1 |
| 2 | 1,004249943 | 1,000532301 |
| 3 | 0,940562638 | 1,028727547 |
| 4 | 0,878340366 | 0,89792911 |
| 5 | 0,728583333 | 0,921268849 |
| 6 | 0,665524853 | 0,837763746 |
| 7 | 0,607194944 | 0,776937019 |
| 8 | 0,573537129 | 0,62958923 |
| 9 | 0,505565675 | 0,631299044 |
| 10 | 0,484645233 | 0,602127507 |
| 11 | 0,45824052 | 0,59537098 |
| 12 | 0,426072125 | 0,563169846 |