**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Кафедра МСС**

**Ульяницкий Владимир Александрович**

**Поддержка многопоточности в с++11: packaged\_task**

Отчет по лабораторной работе № 14

**Преподаватель**

*Кондратьева О. М.*

Минск 2020

1. Рассмотрим задачу нахождения количества максимальных элементов в матрице. Её можно разделить на независимые части следующим образом: находим количество максимумов и сам максимум в каждой строке матрицы, а затем по полученным результатам определяем максимум всей матрицы и находим, сколько раз он встречается. Информационные зависимости между параллельными подзадачами отсутствуют, т. к. в каждой из них один алгоритм выполняется для различных строк матрицы. Для масштабирования подзадач можно разбивать матрицу не на строки, а на блоки из нескольких последовательных строк, оптимальное количество блоков определяется по количеству аппаратно поддерживаемых потоков:

unsigned long const hardware\_threads = std::thread::hardware\_concurrency();

1. Проведем анализ эффективности параллельной реализации в 2-х вариантах: по методу Уильямса и с использованием packaged\_task. Для подсчёта времени выполнения используем возможности библиотеки std::chrono. Для проведения экспериментов сначала реализуем функцию, выполняющую поиск количества максимумов последовательно:

template <typename Iterator, typename T>

struct max\_count\_block

{

void operator()(Iterator first, Iterator last, std::pair<T, int>& result)

{

T max= (\*first)[0];

int count = 0;

std::for\_each(first, last, [&max, &count](const std::vector<T>& row) -> void {

std::for\_each(row.begin(), row.end(), [&max, &count](const T& elem) -> void {

if (elem > max)

{

max = elem;

count = 1;

}

else if (elem == max)

{

count++;

}

});

});

result = std::make\_pair(max, count);

}

};

template <typename Iterator, typename T>

int max\_count(Iterator first, Iterator last)

{

unsigned long const length = std::distance(first, last);

if (!length)

return 0;

std::pair<T, int> result;

max\_count\_block<Iterator, T>()(first, last, result);

return result.second;

}

Структура max\_count\_block может быть затем легко использована в параллельной реализации по Уильямсу, для этого воспользуемся кодом задачи параллельного суммирования с минимальными изменениями:

template <typename Iterator, typename T>

int parallel\_max\_count(Iterator first, Iterator last)

{

unsigned long const length = std::distance(first, last);

if (!length)

return 0;

unsigned long const min\_per\_thread = 25;

unsigned long const max\_threads = (length + min\_per\_thread - 1) / min\_per\_thread;

unsigned long const hardware\_threads = std::thread::hardware\_concurrency();

unsigned long const num\_threads = std::min(hardware\_threads != 0 ? hardware\_threads : 2, max\_threads);

unsigned long const block\_size = length / num\_threads;

std::vector<std::pair<T, int>> results(num\_threads);

std::vector<std::thread> threads(num\_threads - 1);

Iterator block\_start = first;

for (unsigned long i = 0; i < (num\_threads - 1); ++i)

{

Iterator block\_end = block\_start;

std::advance(block\_end, block\_size);

threads[i] = std::thread(max\_count\_block<Iterator, T>(), block\_start, block\_end, std::ref(results[i]));

block\_start = block\_end;

}

max\_count\_block<Iterator, T>()(block\_start, last, results[num\_threads - 1]);

std::for\_each(threads.begin(), threads.end(), std::mem\_fn(&std::thread::join));

T max = results[0].first;

int count = 0;

std::for\_each(results.begin(), results.end(), [&max, &count](const std::pair<T, int>& elem) -> void {

if (elem.first > max)

{

max = elem.first;

count = elem.second;

}

else if (elem.first == max)

{

count += elem.second;

}

});

return count;

}

Для решения задачи с использованием packaged\_task реализуем функцию, которая бы принимала на вход итераторы, указывающие на начало и конец блока векторов, являющихся строками матрицы, и возвращала объект std::pair<T, int>, содержащий максимальный элемент блока и количество раз, которое он встретился:

template <typename Iterator, typename T>

std::pair<T, int> max\_count\_future(Iterator first, Iterator last)

{

T max = (\*first)[0];

int count = 0;

std::for\_each(first, last, [&max, &count](const std::vector<T>& row) -> void {

std::for\_each(row.begin(), row.end(), [&max, &count](const T& elem) -> void {

if (elem > max)

{

max = elem;

count = 1;

}

else if (elem == max)

{

count++;

}

});

});

return std::make\_pair(max, count);

}

Далее реализуем распараллеливание, воспользовавшись примером, приведённым в книге Бьярне Страуструпа «Язык программирования С++. Краткий курс», глава 15.7 «Обмен информацией с заданиями». Для определения количества потоков использован тот же метод, что и в предыдущей реализации, это обеспечит корректность сравнения двух вариантов.

template <typename Iterator, typename T>

int promise\_max\_count(Iterator first, Iterator last)

{

using Task\_type = std::pair<T, int>(Iterator, Iterator);

unsigned long const length = std::distance(first, last);

if (!length)

return 0;

unsigned long const min\_per\_thread = 25;

unsigned long const max\_threads = (length + min\_per\_thread - 1) / min\_per\_thread;

unsigned long const hardware\_threads = std::thread::hardware\_concurrency();

unsigned long const num\_threads = std::min(hardware\_threads != 0 ? hardware\_threads : 2, max\_threads);

unsigned long const block\_size = length / num\_threads;

std::vector<std::future<std::pair<T, int>>> results(num\_threads);

std::vector<std::thread> threads(num\_threads);

Iterator block\_start = first;

for (unsigned long i = 0; i < (num\_threads - 1); ++i)

{

std::packaged\_task<Task\_type> pt{ max\_count\_future<Iterator, T> };

results[i] = pt.get\_future();

Iterator block\_end = block\_start;

std::advance(block\_end, block\_size);

threads[i] = std::thread(move(pt), block\_start, block\_end);

block\_start = block\_end;

}

std::packaged\_task<Task\_type> pt{ max\_count\_future<Iterator, T> };

results[num\_threads - 1] = pt.get\_future();

threads[num\_threads - 1] = std::thread(move(pt), block\_start, last);

std::for\_each(threads.begin(), threads.end(), std::mem\_fn(&std::thread::join));

std::pair<T, int> res = results[0].get();

T max = res.first;

int count = res.second;

std::for\_each(std::next(results.begin()), results.end(), [&max, &count](std::future<std::pair<T, int>>& elem) -> void {

std::pair<T, int> el = elem.get();

if (el.first > max)

{

max = el.first;

count = el.second;

}

else if (el.first == max)

{

count += el.second;

}

});

return count;

}

Наконец выполним тестирование на задаче размерностью 5000\*5000, 10000\*10000 и 15000\*15000 элементов. Корректность работы алгоритмов была предварительно проверена непосредственно на матрице размерностью 10\*10, а также опосредованно с помощью сравнения результатов с последовательным вариантом, с меньшей вероятностью содержащим ошибку, а также сравнением результатов нескольких запусков.

#include <thread>

#include <vector>

#include <algorithm>

#include <numeric>

#include <functional>

#include <future>

int main()

{

std::chrono::high\_resolution\_clock::time\_point start, stop;

int sum;

std::chrono::duration<double> diff;

for (int n : {5'000, 10'000, 15'000})

{

printf("Start init\n");

std::vector<std::vector<int>> matrix(n, std::vector<int>(n));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

matrix[i][j] = (rand() % 1'000) \* (rand() % 1'000);

}

}

printf("End init\n");

start = std::chrono::high\_resolution\_clock::now();

sum = max\_count<std::vector<std::vector<int>>::iterator, int>(matrix.begin(), matrix.end());

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

diff = stop - start;

printf("max count = %i\n", sum);

printf("Time single = %lld, n = %d\n", std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count(), n);

start = std:: chrono::high\_resolution\_clock::now();

sum = parallel\_max\_count<std::vector<std::vector<int>>::iterator, int>(matrix.begin(), matrix.end());

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

diff = stop - start;

printf("max count = %i\n", sum);

printf("Time parallel = %lld, n = %d\n", std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count(), n);

start = std::chrono::high\_resolution\_clock::now();

sum = promise\_max\_count<std::vector<std::vector<int>>::iterator, int>(matrix.begin(), matrix.end());

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

diff = stop - start;

printf("max count = %i\n", sum);

printf("Time promise = %lld, n = %d\n", std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count(), n);

printf("\n\n");

}

return 0;

}

Сначала скомпилируем и запустим в варианте x86, для наибольшего соответствия реальному поведению – в release режиме.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Результаты вычислительных экспериментов | | |  |  |  |
| Число максимумов в матрице | |  |  |  |  |
|  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы | Параллельная программа по Уильямсу | | Параллельная программа packaged\_task | |
| Время выполнения | Ускорение | Время выполнения | Ускорение |
| 5000\*5000 | 63 | 74 | 0,851351351 | 31 | 2,032258065 |
| 10000\*10000 | 266 | 75 | 3,546666667 | 90 | 2,955555556 |
| 15000\*15000 | 579 | 172 | 3,36627907 | 168 | 3,446428571 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы | Параллельная программа по Уильямсу | | Параллельная программа packaged\_task | |
| Время выполнения | Ускорение | Время выполнения | Ускорение |
| 5000\*5000 | 64 | 60 | 1,066666667 | 37 | 1,72972973 |
| 10000\*10000 | 276 | 74 | 3,72972973 | 91 | 3,032967033 |
| 15000\*15000 | 582 | 181 | 3,215469613 | 185 | 3,145945946 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы | Параллельная программа по Уильямсу | | Параллельная программа packaged\_task | |
| Время выполнения | Ускорение | Время выполнения | Ускорение |
| 5000\*5000 | 66 | 59 | 1,118644068 | 28 | 2,357142857 |
| 10000\*10000 | 256 | 83 | 3,084337349 | 78 | 3,282051282 |
| 15000\*15000 | 584 | 185 | 3,156756757 | 166 | 3,518072289 |

Можно видеть, что если на бо́льших размерностях обе реализации обладают примерно одинаковой эффективностью, то на размерности 5000\*5000 только второй вариант достаточно эффективен, в то время как реализация по Уильямсу почти не превосходит последовательную.

Рассмотрим теперь поведение программы при компиляции под архитектуру x64, на которой и выполняется тестирование.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Результаты вычислительных экспериментов | | |  |  |  |
| Число максимумов в матрице | |  |  |  |  |
|  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы | Параллельная программа по Уильямсу | | Параллельная программа packaged\_task | |
| Время выполнения | Ускорение | Время выполнения | Ускорение |
| 5000\*5000 | 79 | 47 | 1,680851064 | 32 | 2,46875 |
| 10000\*10000 | 304 | 91 | 3,340659341 | 79 | 3,848101266 |
| 15000\*15000 | 694 | 315 | 2,203174603 | 171 | 4,058479532 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы | Параллельная программа по Уильямсу | | Параллельная программа packaged\_task | |
| Время выполнения | Ускорение | Время выполнения | Ускорение |
| 5000\*5000 | 77 | 56 | 1,375 | 67 | 1,149253731 |
| 10000\*10000 | 296 | 80 | 3,7 | 87 | 3,402298851 |
| 15000\*15000 | 671 | 178 | 3,769662921 | 192 | 3,494791667 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Размерность задачи | Время выполнения последовательной программы | Параллельная программа по Уильямсу | | Параллельная программа packaged\_task | |
| Время выполнения | Ускорение | Время выполнения | Ускорение |
| 5000\*5000 | 76 | 70 | 1,085714286 | 46 | 1,652173913 |
| 10000\*10000 | 364 | 108 | 3,37037037 | 154 | 2,363636364 |
| 15000\*15000 | 709 | 180 | 3,938888889 | 218 | 3,252293578 |

В этом случае на бо́льших размерностях заметно преимущество реализации по Уильямсу.

Таким образом более высокоуровневая реализация даёт более предсказуемые результаты на задачах различной размерности, к тому же при необходимости адаптировать код для решения другой задачи этот вариант потребует меньших изменений – если у нас есть какая-либо функция, которую мы хотим запустить параллельно для различных данных, в варианте по Уильямсу необходимо реализовать структуру-обёртку, как например для векторного произведения, в то время как в std::packaged\_task можно передавать её напрямую.

Полный код реализации:

#include <thread>

#include <vector>

#include <algorithm>

#include <numeric>

#include <functional>

#include <future>

template <typename Iterator, typename T>

struct max\_count\_block

{

void operator()(Iterator first, Iterator last, std::pair<T, int>& result)

{

T max= (\*first)[0];

int count = 0;

std::for\_each(first, last, [&max, &count](const std::vector<T>& row) -> void {

std::for\_each(row.begin(), row.end(), [&max, &count](const T& elem) -> void {

if (elem > max)

{

max = elem;

count = 1;

}

else if (elem == max)

{

count++;

}

});

});

result = std::make\_pair(max, count);

}

};

template <typename Iterator, typename T>

std::pair<T, int> max\_count\_future(Iterator first, Iterator last)

{

T max = (\*first)[0];

int count = 0;

std::for\_each(first, last, [&max, &count](const std::vector<T>& row) -> void {

std::for\_each(row.begin(), row.end(), [&max, &count](const T& elem) -> void {

if (elem > max)

{

max = elem;

count = 1;

}

else if (elem == max)

{

count++;

}

});

});

return std::make\_pair(max, count);

}

template <typename Iterator, typename T>

int parallel\_max\_count(Iterator first, Iterator last)

{

unsigned long const length = std::distance(first, last);

if (!length)

return 0;

unsigned long const min\_per\_thread = 25;

unsigned long const max\_threads = (length + min\_per\_thread - 1) / min\_per\_thread;

unsigned long const hardware\_threads = std::thread::hardware\_concurrency();

unsigned long const num\_threads = std::min(hardware\_threads != 0 ? hardware\_threads : 2, max\_threads);

unsigned long const block\_size = length / num\_threads;

std::vector<std::pair<T, int>> results(num\_threads);

std::vector<std::thread> threads(num\_threads - 1);

Iterator block\_start = first;

for (unsigned long i = 0; i < (num\_threads - 1); ++i)

{

Iterator block\_end = block\_start;

std::advance(block\_end, block\_size);

threads[i] = std::thread(max\_count\_block<Iterator, T>(), block\_start, block\_end, std::ref(results[i]));

block\_start = block\_end;

}

max\_count\_block<Iterator, T>()(block\_start, last, results[num\_threads - 1]);

std::for\_each(threads.begin(), threads.end(), std::mem\_fn(&std::thread::join));

T max = results[0].first;

int count = 0;

std::for\_each(results.begin(), results.end(), [&max, &count](const std::pair<T, int>& elem) -> void {

if (elem.first > max)

{

max = elem.first;

count = elem.second;

}

else if (elem.first == max)

{

count += elem.second;

}

});

return count;

}

template <typename Iterator, typename T>

int promise\_max\_count(Iterator first, Iterator last)

{

using Task\_type = std::pair<T, int>(Iterator, Iterator);

unsigned long const length = std::distance(first, last);

if (!length)

return 0;

unsigned long const min\_per\_thread = 25;

unsigned long const max\_threads = (length + min\_per\_thread - 1) / min\_per\_thread;

unsigned long const hardware\_threads = std::thread::hardware\_concurrency();

unsigned long const num\_threads = std::min(hardware\_threads != 0 ? hardware\_threads : 2, max\_threads);

unsigned long const block\_size = length / num\_threads;

std::vector<std::future<std::pair<T, int>>> results(num\_threads);

std::vector<std::thread> threads(num\_threads);

Iterator block\_start = first;

for (unsigned long i = 0; i < (num\_threads - 1); ++i)

{

std::packaged\_task<Task\_type> pt{ max\_count\_future<Iterator, T> };

results[i] = pt.get\_future();

Iterator block\_end = block\_start;

std::advance(block\_end, block\_size);

threads[i] = std::thread(move(pt), block\_start, block\_end);

block\_start = block\_end;

}

std::packaged\_task<Task\_type> pt{ max\_count\_future<Iterator, T> };

results[num\_threads - 1] = pt.get\_future();

threads[num\_threads - 1] = std::thread(move(pt), block\_start, last);

std::for\_each(threads.begin(), threads.end(), std::mem\_fn(&std::thread::join));

std::pair<T, int> res = results[0].get();

T max = res.first;

int count = res.second;

std::for\_each(std::next(results.begin()), results.end(), [&max, &count](std::future<std::pair<T, int>>& elem) -> void {

std::pair<T, int> el = elem.get();

if (el.first > max)

{

max = el.first;

count = el.second;

}

else if (el.first == max)

{

count += el.second;

}

});

return count;

}

template <typename Iterator, typename T>

int max\_count(Iterator first, Iterator last)

{

unsigned long const length = std::distance(first, last);

if (!length)

return 0;

std::pair<T, int> result;

max\_count\_block<Iterator, T>()(first, last, result);

return result.second;

}

int main()

{

std::chrono::high\_resolution\_clock::time\_point start, stop;

int sum;

std::chrono::duration<double> diff;

for (int n : {5'000, 10'000, 15'000})

{

printf("Start init\n");

std::vector<std::vector<int>> matrix(n, std::vector<int>(n));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

matrix[i][j] = (rand() % 1'000) \* (rand() % 1'000);

}

}

printf("End init\n");

start = std::chrono::high\_resolution\_clock::now();

sum = max\_count<std::vector<std::vector<int>>::iterator, int>(matrix.begin(), matrix.end());

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

diff = stop - start;

printf("max count = %i\n", sum);

printf("Time single = %lld, n = %d\n", std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count(), n);

start = std:: chrono::high\_resolution\_clock::now();

sum = parallel\_max\_count<std::vector<std::vector<int>>::iterator, int>(matrix.begin(), matrix.end());

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

diff = stop - start;

printf("max count = %i\n", sum);

printf("Time parallel = %lld, n = %d\n", std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count(), n);

start = std::chrono::high\_resolution\_clock::now();

sum = promise\_max\_count<std::vector<std::vector<int>>::iterator, int>(matrix.begin(), matrix.end());

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

diff = stop - start;

printf("max count = %i\n", sum);

printf("Time promise = %lld, n = %d\n", std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count(), n);

printf("\n\n");

}

return 0;

}