Петров Андрей, 13 группа 3 курс

15. 7. Обмен информацией с заданиями

Стандартная библиотека предоставляет несколько средств, позволяющих программистам работать на концептуальном уровне заданий (потенциально работающих параллельно), а не непосредственно на низком уровне потоков и блокировок:

- ***future*** и ***promise*** для возврата значения из задания, запущенного в отдельном потоке;
- ***packaged_task*** для помощи в запуске задач и подключении механизмов для возврата результата;
- ***async()*** для запуска заданий способом, очень похожим на вызов функции.

Задание

Задача

Поиск значения максимального элемента

вектора (в аудитории)

матрицы (дома)

Решение

Использовать packaged_task

1. Создадим последовательную программу

```
int vector_max(vector<int> &vec)
{
   return *max_element(vec.begin(), vec.end());
}

int matrix_max_seq(vector<vector<int>> &matrix) {
    int max_value = 0;
    int tmp_value = 0;
    for(auto &row : matrix) {
        tmp_value = vector_max(row);
    }
    if(tmp_value > max_value) max_value = tmp_value;
    return max_value;
}

int main(int argc, char **argv)
{
    LARGE_INTEGER liFrequency, liStartTime, liFinishTime;
    double dElapsedTime;
    QueryPerformanceFrequency(&liFrequency);

   vector<vector<int>> matrix(NUMBER_ROWS, vector<int>(NUMBER_COLUMNS));
    fill_matrix(matrix, NUMBER_ROWS, NUMBER_COLUMNS);
```

```
    QueryPerformanceCounter(&liStartTime);
    int total_max = matrix_max_seq(matrix);
    QueryPerformanceCounter(&liFinishTime);

    dElapsedTime = 1000.0 * (liFinishTime.QuadPart - liStartTime.QuadPart) /
liFrequency.QuadPart;
    printf("Time:  %f ms\n", dElapsedTime);
    printf("Total max: %d\n", total_max);
   return 0;
}
```

2. Создадим параллельную программу

```
int vector_max(vector<int> &vec)
{
   return *max_element(vec.begin(), vec.end());
}

int matrix_row_max(vector<vector<int>> &matrix, int thread_index, int
thread_count)
{
   int n = matrix.size();
   int r = n % thread_count;
   int start = min(thread_index, r) + (n / thread_count) * thread_index;
   int end = min(thread_index + 1, r) + (n / thread_count) * (thread_index +
1);

   vector<int> maximums;
   for (thread_index = start; thread_index < end; ++thread_index) {
      maximums.emplace_back(vector_max(matrix[thread_index]));
   }
   return vector_max(maximums);
}

int main(int argc, char **argv)
{
    LARGE_INTEGER liFrequency, liStartTime, liFinishTime;
    double dElapsedTime;
    QueryPerformanceFrequency(&liFrequency);

   vector<vector<int>> matrix(NUMBER_ROWS, vector<int>(NUMBER_COLUMNS));
    fill_matrix(matrix, NUMBER_ROWS, NUMBER_COLUMNS);

    QueryPerformanceCounter(&liStartTime);
    vector<int> maximums(THREAD_COUNT);
    vector<future<int>> futures;
    vector<thread> threads;

   for (auto i = 0; i < THREAD_COUNT; ++i) {
      auto task = packaged_task<int(vector<vector<int>> &, int,
int)>(matrix_row_max);
      futures.emplace_back(task.get_future());
```

```
        threads.emplace_back(move(task), ref(matrix), i, THREAD_COUNT);
    }

    for (auto i = 0; i < THREAD_COUNT; ++i) {
        maximums[i] = futures[i].get();
        threads[i].join();
    }
    int total_max = vector_max(maximums);
     QueryPerformanceCounter(&liFinishTime);

     dElapsedTime = 1000.0 * (liFinishTime.QuadPart - liStartTime.QuadPart) /
liFrequency.QuadPart;
     printf("Time:  %f ms\n", dElapsedTime);
     printf("Total max: %d\n", total_max);
    return 0;
}
```

Произведем вычислительные эксперименты и составим таблицу

Размерность матрицы	Последовательная программа	Параллельная программа	Ускорение
100 x 100	0.140100	1.616700	11.5396146
1 000 x 1 000	10.062500	3.916900	0.389257
10 000 x 10 000	715.511200	275.881900	0.38557314