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

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №4:**

**«Технология OpenMP. Особенности настройки»**

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 - CLion 2024.2.1, G++ 13.1, OpenMP 201511

*Описание рассмотренных в работе опций и директив*:

OpenMP version: 201511

Year: 2015

Num of processors: 12

Max num of threads: 12

Dynamic threading is off

Timer resolution: 0.001 (in seconds)

Nested parallelism is off

Max active levels: 1

Schedule: dynamic

Chunk size: 1

Чтобы узнать версию и дату принятия стандарта OpenMP, был использован макрос \_OPENMP. Первые 4 цифры в нем указывают год принятия стандарта.

С помощью функций omp\_get\_num\_procs() и omp\_get\_max\_threads() были получены числа доступных процессоров и максимального числа потоков.

Опция dynamic управляет динамическим распределением потоков, что позволяет автоматически изменять количество потоков во время выполнения программы. С помощью функции omp\_get\_dynamic() получено состояние этой опции.

Для определения разрешения таймера была использована функция omp\_get\_wtick(), которая возвращает время в секундах между двумя соседними измерениями таймера.

С помощью функций omp\_get\_nested() и omp\_get\_max\_active\_levels() были получены значения вложенности параллельных областей.

Для получения информации о текущем способе распределения нагрузки была использована функция omp\_get\_schedule().

* omp\_sched\_static: Итерации делятся на фиксированные части и распределяются между потоками до начала выполнения функции. Если chunk\_size не указан, то весь диапазон делится на равные части.
* omp\_sched\_dynamic: Итерации разделяются на части, которые динамически распределяются между потоками по мере выполнения ими текущей задачи.
* omp\_sched\_guided: Итерации сначала распределяются на большие части, которые уменьшаются в размере по мере выполнения.
* omp\_sched\_auto: Планирование определяется средой выполнения OpenMP, и поведение может меняться в зависимости от реализации OpenMP.

chunk\_size определяет количество итераций, которые обрабатываются одним потоком за один раз.

*Пример алгоритма с использованием omp\_set\_lock():*

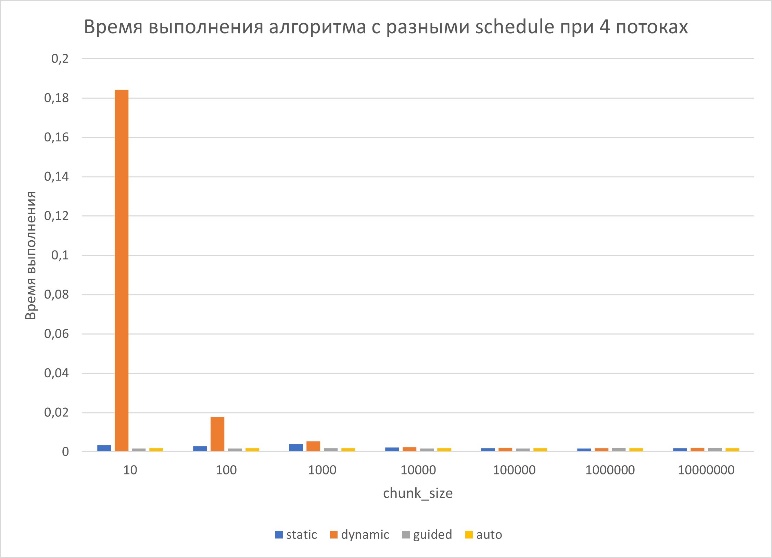
void algo\_with\_lock(){  
 std::mt19937\_64 gen(SEED);  
 std::uniform\_int\_distribution dist(1, 6);  
 std::vector<int> array;  
 omp\_lock\_t lock;  
  
 omp\_init\_lock(&lock);  
  
#pragma omp parallel num\_threads(12) shared(gen, dist, array)  
 {  
 const int thread\_num = omp\_get\_thread\_num();  
 const int num = dist(gen);  
  
 omp\_set\_lock(&lock);  
 const bool isInArray = std::ranges::find(array, num) != array.end();  
  
 if (!isInArray){  
 cout << "Thread " << thread\_num << " adds number " << num << endl;  
 array.push\_back(num);  
 }  
 omp\_unset\_lock(&lock);  
 }  
 omp\_destroy\_lock(&lock);  
}

В данной функции несколько потоков могут одновременно проверять и изменять общий вектор array. Без блокировки возможны ситуации, когда два потока одновременно проверяют array и добавляют значение, что приведет к ошибке в многопоточном доступе:

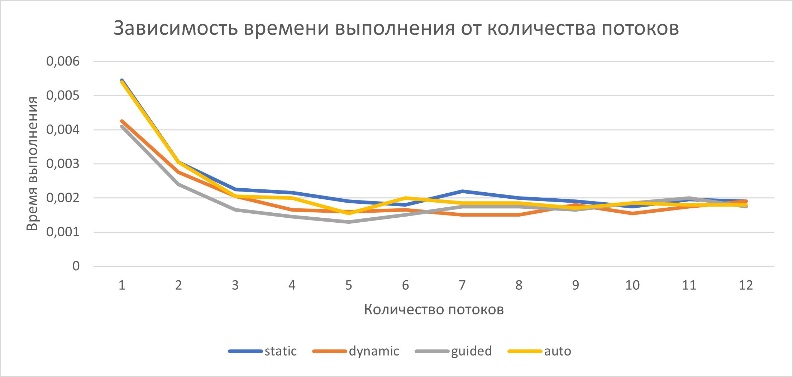
|  |  |
| --- | --- |
|  |  |
| Вывод функции без использования lock | Вывод функции с использованием lock |

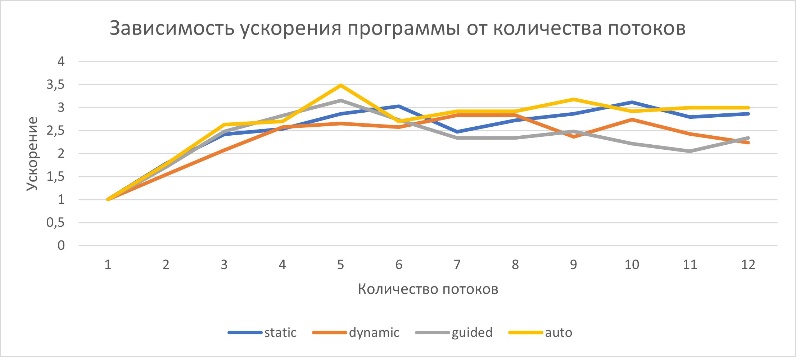
*Графики выполнения программы:*

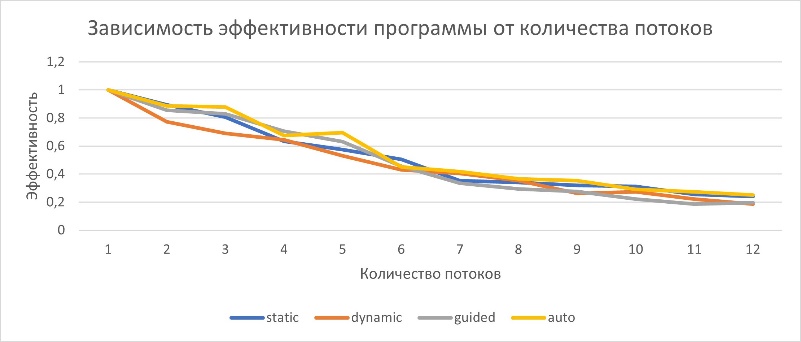
Использовался алгоритм нахождения индекса элемента по заданному значению из 2 лабораторной работы (опция auto не требует chunk\_size, однако его значения были добавлены в каждый столбец для наглядности).



Далее использовался chunk\_size = 10000, т.к. при нем показывались примерно равные результаты на 4 потоках (для auto он был убран).







*Заключение:*

В ходе данной лабораторной работы были изучены различные функции и директивы технологии OpenMP для организации параллельных вычислений, в том числе на примере функции поиска заданного элемента. Программа была запущена при разном количестве потоков и разных параметрах schedule и chunk\_size, что позволило оценить влияние распараллеливания на производительность и время выполнения алгоритма.

В ходе запуска программы с различными режимами планирования (static, dynamic, guided, auto) и разными значениями chunk\_size были выявлены следующие особенности:

1. Static:

Режим static показал стабильные и достаточно хорошие результаты на всех тестах. Этот метод распределяет задачи равномерно до начала выполнения, что минимизирует накладные расходы, и подходит для задачи поиска элемента, т.к. сложность выполнения для всех потоков одинакова. Эффективность static планирования была высокой при всех количествах потоков.

1. Dynamic:

Режим dynamic лучше подходит для ситуаций, когда время выполнения итераций сильно варьируется, поскольку он распределяет задания потокам по мере их выполнения, а в данной задаче этого не требуется. На малом chunk\_size этот режим показал меньшую производительность из-за накладных расходов на управление задачами (overhead), но на большем размере чанков его эффективность увеличивается.

1. Guided:

Режим guided изначально выделяет большие задачи, которые уменьшаются по мере выполнения, что помогает распределять нагрузку эффективнее, чем static, но с меньшими накладными расходами, чем dynamic. Тем не менее, на большом количестве потоков (6-12) этот режим показал худшие результаты, возможно из-за больших накладных расходов на распределение уменьшающихся по размеру задач.

1. Auto:

Режим auto позволяет среде выполнения OpenMP самостоятельно определять наиболее эффективное распределение итераций, что дает сбалансированные результаты. Auto показал средние результаты, не доминируя ни в одной категории, но предоставляя стабильную производительность на всех уровнях параллелизма.

*Приложение:*

1. Исходный код программы для получения значений различных настроек OpenMP:

#include <iostream>  
#include <omp.h>  
#include <random>  
#include <algorithm>  
#include <vector>  
  
#define SEED 920215  
  
using std::cout, std::endl, std::vector;  
  
void algo\_with\_lock(){  
 std::mt19937\_64 gen(SEED);  
 std::uniform\_int\_distribution dist(1, 6);  
 std::vector<int> array;  
 omp\_lock\_t lock;  
  
 omp\_init\_lock(&lock);  
  
#pragma omp parallel num\_threads(12) shared(gen, dist, array)  
 {  
 const int thread\_num = omp\_get\_thread\_num();  
 const int num = dist(gen);  
  
 omp\_set\_lock(&lock);  
 const bool isInArray = std::ranges::find(array, num) != array.end();  
  
 if (!isInArray){  
 cout << "Thread " << thread\_num << " adds number " << num << endl;  
 array.push\_back(num);  
 }  
 omp\_unset\_lock(&lock);  
 }  
 omp\_destroy\_lock(&lock);  
}  
  
int main(){  
 // 1  
 cout << "OpenMP version: " << \_OPENMP << endl;  
 cout << "Year: " << \_OPENMP / 100 << endl;  
  
 // 2  
 const int num\_procs = omp\_get\_num\_procs();  
 const int max\_threads = omp\_get\_max\_threads();  
  
 cout << "Num of processors: " << num\_procs << endl;  
 cout << "Max num of threads: " << max\_threads << endl;  
  
 // 3  
 const int is\_dynamic = omp\_get\_dynamic();  
 cout << "Dynamic threading is " << (is\_dynamic ? "on" : "off") << endl;  
  
 // 4  
 const double tick = omp\_get\_wtick();  
 cout << "Timer resolution: " << tick << " (in seconds)" << endl;  
  
 // 5  
 const int nested = omp\_get\_nested();  
 const int max\_levels = omp\_get\_max\_active\_levels();  
  
 cout << "Nested parallelism is " << (nested ? "on" : "off") << endl;  
 cout << "Max active levels: " << max\_levels << endl;  
  
 // 6  
 omp\_sched\_t schedule;  
 int chunk\_size;  
  
 omp\_get\_schedule(&schedule, &chunk\_size);  
  
 cout << "Schedule: ";  
 switch (schedule){  
 case omp\_sched\_static:  
 cout << "static";  
 break;  
 case omp\_sched\_dynamic:  
 cout << "dynamic";  
 break;  
 case omp\_sched\_guided:  
 cout << "guided";  
 break;  
 case omp\_sched\_auto:  
 cout << "auto";  
 break;  
 default:  
 cout << "unknown";  
 }  
 cout << endl;  
 cout << "Chunk size: " << chunk\_size << endl;  
  
 // 7  
 algo\_with\_lock();  
  
 return 0;  
}

1. Исходный код программы для разных типов разделения нагрузки и размеров фрагмента (опция schedule директивы parallel) – поиск заданного элемента из 2 лабораторной работы.

#include <stdio.h>  
#include <stdlib.h>  
#include <omp.h>  
  
void initialize\_array(const int random\_seed, int \*array, const int count){  
 srand(random\_seed);  
  
 for (int i = 0; i < count; i++){  
 array[i] = rand() \* rand() \* (i % 2 == 0 ? 2 : 3);  
 }  
}  
  
double find\_element\_index(const int \*array, const int count, const int target, const int threads){  
 int num;  
 int index = -1;  
 int elementIsFound = 0;  
 double time\_spent = 0;  
 double begin = omp\_get\_wtime();  
  
#pragma omp parallel num\_threads(threads) shared(array, count, target, index, elementIsFound) private(num) default(shared)  
 {  
 num = omp\_get\_thread\_num();  
#pragma omp for schedule(auto)  
 for (int i = 0; i < count; i++){  
 if (elementIsFound){  
 i = count;  
 continue;  
 }  
 if (array[i] == target){  
 index = i;  
 printf("%d\n", index);  
 elementIsFound = 1;  
 }  
 }  
 }  
 double end = omp\_get\_wtime();  
 time\_spent = end - begin;  
  
 return time\_spent;  
}  
  
void time\_algorithm(const int count, int \*array, int random\_seed, const int threads, const int runs\_num,  
 double time\_spent, const int target){  
 for (int t = 1; t <= 12; t++){  
 for (int i = 0; i < runs\_num; i++){  
 initialize\_array(random\_seed, array, count);  
 random\_seed += 12345;  
 time\_spent += find\_element\_index(array, count, target, t);  
 }  
 printf("%lf\n", time\_spent / runs\_num);  
 time\_spent = 0;  
 random\_seed = 920214;  
 }  
}  
  
int main(int argc, char \*\*argv){  
 const int count = 10000000; ///< Number of array elements  
 const int random\_seed = 920214; ///< RNG seed  
 const int target = INT\_MAX; ///< Number to look for  
 const int threads = 12;  
 int \*array = malloc(count \* sizeof(int));  
 int index = -1; ///< The index of the element we need  
 const int runs\_num = 20;  
 const double time\_spent = 0;  
  
 printf("OpenMP: %d\n", \_OPENMP);  
  
 time\_algorithm(count, array, random\_seed, threads, runs\_num, time\_spent, target);  
  
 return 0;  
}

1. Таблица времени работы программы с разными schedule

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения с разными schedule | | | | |
| **chunk\_size** | **static** | **dynamic** | **guided** | **auto** |
| 10 | 0,00345 | 0,1843 | 0,00175 | 0,00195 |
| 100 | 0,00295 | 0,0178 | 0,0018 | 0,00195 |
| 1000 | 0,00395 | 0,0054 | 0,0019 | 0,00195 |
| 10000 | 0,00225 | 0,0025 | 0,0017 | 0,00195 |
| 100000 | 0,00185 | 0,0021 | 0,0017 | 0,00195 |
| 1000000 | 0,00175 | 0,0019 | 0,00205 | 0,00195 |
| 10000000 | 0,00185 | 0,0021 | 0,0021 | 0,00195 |

1. Таблица времени работы программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения при разном количестве потоков | | | | |
| **Количество потоков** | **static** | **dynamic** | **guided** | **auto** |
| 1 | 0,00545 | 0,00425 | 0,0041 | 0,0054 |
| 2 | 0,00305 | 0,00275 | 0,0024 | 0,00305 |
| 3 | 0,00225 | 0,00205 | 0,00165 | 0,00205 |
| 4 | 0,00215 | 0,00165 | 0,00145 | 0,002 |
| 5 | 0,0019 | 0,0016 | 0,0013 | 0,00155 |
| 6 | 0,0018 | 0,00165 | 0,0015 | 0,002 |
| 7 | 0,0022 | 0,0015 | 0,00175 | 0,00185 |
| 8 | 0,002 | 0,0015 | 0,00175 | 0,00185 |
| 9 | 0,0019 | 0,0018 | 0,00165 | 0,0017 |
| 10 | 0,00175 | 0,00155 | 0,00185 | 0,00185 |
| 11 | 0,00195 | 0,00175 | 0,002 | 0,0018 |
| 12 | 0,0019 | 0,0019 | 0,00175 | 0,0018 |

1. Таблица ускорения программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ускорение программы | | | | |
| **Количество потоков** | **static** | **dynamic** | **guided** | **auto** |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1,786885 | 1,5454545 | 1,708333 | 1,770492 |
| 3 | 2,422222 | 2,0731707 | 2,484848 | 2,634146 |
| 4 | 2,534884 | 2,5757576 | 2,827586 | 2,7 |
| 5 | 2,868421 | 2,65625 | 3,153846 | 3,483871 |
| 6 | 3,027778 | 2,5757576 | 2,733333 | 2,7 |
| 7 | 2,477273 | 2,8333333 | 2,342857 | 2,918919 |
| 8 | 2,725 | 2,8333333 | 2,342857 | 2,918919 |
| 9 | 2,868421 | 2,3611111 | 2,484848 | 3,176471 |
| 10 | 3,114286 | 2,7419355 | 2,216216 | 2,918919 |
| 11 | 2,794872 | 2,4285714 | 2,05 | 3 |
| 12 | 2,868421 | 2,2368421 | 2,342857 | 3 |

1. Таблица эффективности программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ускорение программы | | | | |
| **Количество потоков** | **static** | **dynamic** | **guided** | **auto** |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0,893443 | 0,7727273 | 0,854167 | 0,885246 |
| 3 | 0,807407 | 0,6910569 | 0,828283 | 0,878049 |
| 4 | 0,633721 | 0,6439394 | 0,706897 | 0,675 |
| 5 | 0,573684 | 0,53125 | 0,630769 | 0,696774 |
| 6 | 0,50463 | 0,4292929 | 0,455556 | 0,45 |
| 7 | 0,353896 | 0,4047619 | 0,334694 | 0,416988 |
| 8 | 0,340625 | 0,3541667 | 0,292857 | 0,364865 |
| 9 | 0,318713 | 0,2623457 | 0,276094 | 0,352941 |
| 10 | 0,311429 | 0,2741935 | 0,221622 | 0,291892 |
| 11 | 0,254079 | 0,2207792 | 0,186364 | 0,272727 |
| 12 | 0,239035 | 0,1864035 | 0,195238 | 0,25 |