МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

СЕРГИЕНКО ЛЕВ ЭДУАРДОВИЧ

Отчет по
Лабораторная работа 9
Технология OpenMP: распараллеливание циклов

Преподаватель

Кондратьева О.М.

2025

Задание 1

Текст программы

```
#include <iostream>
#include <omp.h>

int main()
{
#pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();

#pragma omp critical
        {
            std::cout << "Привет от потока "
                      << tid
                      << " из "
                      << nthreads
                      << std::endl;
        }
    }

    return 0;
}
```

Окно работы программы

```
lev@redmibook-15 /m/c/U/l/D/B/3/p/lab9 (main)> make run1
g++ -std=c++11 -fopenmp task1.cpp -o task1
./task1
Привет от потока 15 из 16
Привет от потока 9 из 16
Привет от потока 1 из 16
Привет от потока 3 из 16
Привет от потока 12 из 16
Привет от потока 2 из 16
Привет от потока 5 из 16
Привет от потока 4 из 16
Привет от потока 6 из 16
Привет от потока 7 из 16
Привет от потока 8 из 16
Привет от потока 10 из 16
Привет от потока 11 из 16
Привет от потока 13 из 16
Привет от потока 14 из 16
Привет от потока 0 из 16
```

Задание 2

Текст программы

```
#include <iostream>
#include <omp.h>

int main()
{
    const int N = 16;

    omp_set_num_threads(4);

    int count[32] = {0};

#pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        int tid = omp_get_thread_num();
#pragma omp atomic
        ++count[tid];

#pragma omp critical
        {
            std::cout << "i=" << i
                      << " выполняет поток " << tid
                      << std::endl;
        }
    }

    std::cout << "\nИтераций по потокам:\n";
    for (int t = 0; t < omp_get_max_threads(); ++t)
    {
        std::cout << "Поток " << t << ": "
                  << count[t] << " итераций\n";
    }

    return 0;
}


#include <iostream>
#include <cmath>
#include <iomanip>
#include <omp.h>

int main()
{
    constexpr double PI = 3.141592653589;
```

```
    const int sizes[] = {256, 1024, 4096};
    const int threads[] = {1, 2, 4, 8, 16};

    std::cout << " size | threads |   time (s)   \n";
    std::cout << "------+---------+--------------\n";

    for (int sz : sizes)
    {
        for (int nt : threads)
        {
            omp_set_num_threads(nt);
            double *sinTable = new double[sz];

            double t0 = omp_get_wtime();
#pragma omp parallel for
            for (int n = 0; n < sz; ++n)
            {
                sinTable[n] = std::sin(2 * PI * n / sz);
            }
            double t1 = omp_get_wtime();

            std::cout
                << std::setw(5) << sz << " | "
                << std::setw(7) << nt << " | "
                << std::fixed
                << std::setprecision(6)
                << (t1 - t0) << "\n";

            delete[] sinTable;
        }
    }

    return 0;
}
```

Таблицы с результатами экспериментов

```
i=0 выполняет поток 0
i=8 выполняет поток 2
i=1 выполняет поток 0
i=2 выполняет поток 0
i=3 выполняет поток 0
i=12 выполняет поток 3
i=4 выполняет поток 1
i=13 выполняет поток 3
i=14 выполняет поток 3
i=15 выполняет поток 3
i=5 выполняет поток 1
i=6 выполняет поток 1
i=7 выполняет поток 1
i=9 выполняет поток 2
i=10 выполняет поток 2
i=11 выполняет поток 2

Итераций по потокам:
Поток 0: 4 итераций
Поток 1: 4 итераций
Поток 2: 4 итераций
Поток 3: 4 итераций
```

size	threads	time (s)
1024	1	0.000032
1024	2	0.000088
1024	4	0.000110
1024	8	0.000252
1024	16	0.002590
4096	1	0.000069
4096	2	0.000030
4096	4	0.000417
4096	8	0.000171
4096	16	0.003978
16384	1	0.000323
16384	2	0.000142
16384	4	0.000434
16384	8	0.000300
16384	16	0.003918
32768	1	0.000910
32768	2	0.000505
32768	4	0.000769
32768	8	0.000300
32768	16	0.003623

Выводы

Распределение итераций

1. OpenMP по умолчанию использует статический режим, при котором диапазон 0…N-1 равномерно разбивается на блоки подряд идущих итераций размера $\lceil N/T \rceil$ (или $\lfloor N/T \rfloor$).
2. В эксперименте при N=16 и T=4 каждый из четырёх потоков получил ровно по 4 итерации.
3. Порядок строк в выводе (i=8 первым, потом i=4 и т. д.) определяется скоростью захода потоков в критическую секцию и не гарантирует упорядоченный вывод по i.

Влияние накладных расходов

Для маленького объёма работы (size = 1024) параллельный запуск во всех конфигурациях оказался медленнее однопоточного:
NT=1: 0.000032 с
NT=2: 0.000088 с

NT=4: 0.000110 с
NT=8: 0.000252 с
NT=16: 0.002590 с

Наладка потоков, синхронизация и перераспределение итераций «съедают» выигрыш при малом объёме, поэтому при N≤4000 даже два потока часто проигрывают одному.

Ускорение с ростом размера данных

1. При size = 4096 заметен выигрыш только для 2 потоков: 0.000030 с vs 0.000069 с (однопоточный).
2. При size = 16384 также лучшим оказался двухпоточный режим: 0.000142 с vs 0.000323 с.
3. Для самого большого теста (size = 32768) максимальное ускорение (0.000300 с) достигнуто при 8 потоках.

Задание 3

Выводы

В директиве

#pragma omp parallel for schedule(<kind>[,chunk_size])

параллельный for разбивается на «закрепленные» участки (chunks) по правилам выбранного алгоритма:

- **static**
  Итерации делятся на почти равные блоки заранее: каждый поток получает одну (или несколько, если указан chunk_size) подряд идущих итераций. Низкие накладные расходы, но при неравномерных по времени итерациях, возможен дисбаланс.

  Лучшее быстродействие при **равномерно** нагруженных итерациях.

  Плохо подходит, если время выполнения итераций сильно варьируется: одни потоки «зависают» на тяжёлых итерациях, другие простаивают.

- **dynamic**
  Каждому потоку по очереди выдаётся блок (по умолчанию размер 1, либо chunk_size) из очереди до её опустения. При внезапно «тяжёлых» итерациях даёт лучший баланс, но с более высокими накладными расходами на выдачу блоков.

  Хороший баланс даже при сильно разнородных итерациях, каждый поток берёт новую работу по факту готовности.

  Высокие накладные расходы на синхронизацию и раздачу мелких задач, особенно при малом объёме работы.

- **guided**
  Похож на dynamic, но размер блока уменьшается по мере «опустения» итераций: сначала большие участки (ceil(N/threads)), потом всё меньше, пока не достигнет chunk_size (по умолчанию 1). Компромисс между балансом и накладными расходами.

  Снижает общий overhead динамической схемы за счёт выдачи больших блоков в начале и уменьшения размера блока по ходу.

  Идеален, когда изначально много работы, но последние несколько итераций могут быть тяжёлыми или лёгкими, сохраняет баланс без чрезмерных синхронизаций.

Задание 4

Текст программы

```
#include <iostream>
#include <omp.h>

int main()
{
    const int value = 2;
    const int n = 1000000;
    int counter = 0;

#pragma omp parallel for
    for (int i = 0; i < n; ++i)
    {
// закомментировать pragma, чтобы увидеть погрешность
#pragma omp atomic
        counter += value;
    }

    std::cout << "counter = " << counter << std::endl;
    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <random>
#include <limits>
#include <omp.h>
#include <iomanip>

int main()
{
    const int sizes[] = {1000000, 5000000, 10000000, 100000000};
    const int threads[] = {1, 2, 4, 8, 16};

    std::mt19937 gen(42);
    std::uniform_real_distribution<> dist(-1e6, 1e6);

    std::cout << " size   | threads |   time (s)\n";
    std::cout << "--------+---------+------------\n";

    for (int sz : sizes)
```

```
    {
        std::vector<double> a(sz);
        for (int i = 0; i < sz; ++i)
            a[i] = dist(gen);

        for (int nt : threads)
        {
            omp_set_num_threads(nt);

            double t0 = omp_get_wtime();

            double global_min = std::numeric_limits<double>::infinity();
#pragma omp parallel for
            for (int i = 0; i < sz; ++i)
            {
                if (a[i] < global_min)
                {
#pragma omp critical
                    {
                        if (a[i] < global_min)
                            global_min = a[i];
                    }
                }
            }

            double t1 = omp_get_wtime();
            double dt = t1 - t0;

            std::cout
                << std::setw(7) << sz << " | "
                << std::setw(7) << nt << " | "
                << std::fixed << std::setprecision(6)
                << dt << "\n";
        }
    }

    return 0;
}
```

Таблицы с результатами экспериментов

Размерность задачи	Время выполнения последовательной, с	2 потока			4 потока			8 потока			16 потока		
		Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность
1 000 000	0,002112	0,001476	1,430894309	0,7154471545	0,000939	2,249201278	0,5623003195	0,000986	2,141987783	0,2677484787	0,004085	0,5170134639	0,03231334149
5 000 000	0,009512	0,005281	1,801174702	0,900587701	0,00352	2,702272727	0,6755681818	0,002585	3,679690522	0,4599613153	0,007334	1,296973002	0,08106081265
10 000 000	0,018817	0,010188	1,846976835	0,9234884177	0,005468	3,441294806	0,8603237015	0,007981	2,357724596	0,2947155745	0,005941	3,1673119	0,1979569938
100 000 000	0,188862	0,122711	1,539079626	0,7695398131	0,057357	3,292745436	0,8231863591	0,05123	3,686550849	0,4608188561	0,029462	6,410359107	0,4006474442

Задание 5

Тексты программ

```
#include <iostream>
#include <omp.h>

// Факториал с reduction
long long factorial_reduction(int number)
{
    long long fac = 1;
#pragma omp parallel for reduction(* : fac)
    for (int n = 2; n <= number; ++n)
    {
        fac *= n;
    }
    return fac;
}

// Факториал с atomic
long long factorial_atomic(int number)
{
    long long fac = 1;
#pragma omp parallel for
    for (int n = 2; n <= number; ++n)
    {
#pragma omp atomic
        fac *= n;
    }
    return fac;
}

int main()
```

```
{
    const int Ns[] = {1000000, 10000000, 100000000};
    const int threads[] = {1, 2, 4, 8};

    std::cout << "Threads | reduction time (s) | atomic time (s)\n";
    for (int t : threads)
    {
        omp_set_num_threads(t);

        for (int N : Ns)
        {
            double t0 = omp_get_wtime();
            volatile long long r1 = factorial_reduction(N);
            double tr = omp_get_wtime() - t0;

            double t1 = omp_get_wtime();
            volatile long long r2 = factorial_atomic(N);
            double ta = omp_get_wtime() - t1;

            std::cout
                << t << "\t| "
                << tr << "\t\t| "
                << ta << "\n";
        }
        std::cout << "-----------------------------------------\n";
    }
    return 0;
}

////////////////////////////////////


#include <iostream>
#include <iomanip>
#include <cmath>
#include <omp.h>

double pi_serial(int N)
{
    double sum = 0.0;
    for (int n = 0; n < N; ++n)
        sum += ((n % 2 == 0 ? 1.0 : -1.0) / (2 * n + 1));
    return 4.0 * sum;
}

double pi_reduction(int N)
{
    double sum = 0.0;
#pragma omp parallel for reduction(+ : sum)
    for (int n = 0; n < N; ++n)
        sum += ((n % 2 == 0 ? 1.0 : -1.0) / (2 * n + 1));
```

```
    return 4.0 * sum;
}

double pi_atomic(int N)
{
    double sum = 0.0;
#pragma omp parallel for
    for (int n = 0; n < N; ++n)
    {
        double term = (n % 2 == 0 ? 1.0 : -1.0) / (2 * n + 1);
#pragma omp atomic
        sum += term;
    }
    return 4.0 * sum;
}

double pi_critical(int N)
{
    double sum = 0.0;
#pragma omp parallel for
    for (int n = 0; n < N; ++n)
    {
        double term = (n % 2 == 0 ? 1.0 : -1.0) / (2 * n + 1);
#pragma omp critical
        sum += term;
    }
    return 4.0 * sum;
}

int main()
{
    const int Ns[] = {1000000, 5000000, 10000000};
    const int threads[] = {2, 4, 8};
    const double PI_REF = std::acos(-1.0);

    std::cout << std::fixed << std::setprecision(6);
    std::cout << "Method     |      N | Threads |   Time(s) |    Error\n";
    std::cout << "-----------------------------------------------------------\n";

    for (int N : Ns)
    {
        // serial
        double t0 = omp_get_wtime();
        double ps = pi_serial(N);
        double ts = omp_get_wtime() - t0;
        double es = std::fabs(ps - PI_REF);
        std::cout
            << std::setw(10) << "serial"
            << " | " << std::setw(7) << N
            << " | " << std::setw(7) << 1
            << " | " << std::setw(8) << ts
```

```
            << " | " << std::setw(8) << es
            << "\n";

        for (int t : threads)
        {
            omp_set_num_threads(t);

            // reduction
            t0 = omp_get_wtime();
            double pr = pi_reduction(N);
            double tr = omp_get_wtime() - t0;
            double er = std::fabs(pr - PI_REF);
            std::cout
                << std::setw(10) << "reduction"
                << " | " << std::setw(7) << N
                << " | " << std::setw(7) << t
                << " | " << std::setw(8) << tr
                << " | " << std::setw(8) << er
                << "\n";

            // atomic
            t0 = omp_get_wtime();
            double pa = pi_atomic(N);
            double ta = omp_get_wtime() - t0;
            double ea = std::fabs(pa - PI_REF);
            std::cout
                << std::setw(10) << "atomic"
                << " | " << std::setw(7) << N
                << " | " << std::setw(7) << t
                << " | " << std::setw(8) << ta
                << " | " << std::setw(8) << ea
                << "\n";

            // critical
            t0 = omp_get_wtime();
            double pc = pi_critical(N);
            double tc = omp_get_wtime() - t0;
            double ec = std::fabs(pc - PI_REF);
            std::cout
                << std::setw(10) << "critical"
                << " | " << std::setw(7) << N
                << " | " << std::setw(7) << t
                << " | " << std::setw(8) << tc
                << " | " << std::setw(8) << ec
                << "\n";
        }
        std::cout <<
"-----------------------------------------------------\n";
    }

    return 0;
```

```
}

////////////////////////////////////


#include <iostream>
#include <iomanip>
#include <vector>
#include <random>
#include <omp.h>

double dot_serial(const std::vector<double> &A, const std::vector<double> &B)
{
    double sum = 0.0;
    size_t M = A.size();
    for (size_t i = 0; i < M; ++i)
        sum += A[i] * B[i];
    return sum;
}

double dot_reduction(const std::vector<double> &A, const std::vector<double> &B)
{
    double sum = 0.0;
    size_t M = A.size();
#pragma omp parallel for reduction(+ : sum)
    for (size_t i = 0; i < M; ++i)
        sum += A[i] * B[i];
    return sum;
}

double dot_atomic(const std::vector<double> &A, const std::vector<double> &B)
{
    double sum = 0.0;
    size_t M = A.size();
#pragma omp parallel for
    for (size_t i = 0; i < M; ++i)
    {
#pragma omp atomic
        sum += A[i] * B[i];
    }
    return sum;
}

double dot_manual(const std::vector<double> &A, const std::vector<double> &B)
{
    double global_sum = 0.0;
    size_t M = A.size();
#pragma omp parallel
    {
```

```
        double local_sum = 0.0;
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        size_t chunk = M / nthreads;
        size_t start = tid * chunk;
        size_t end = (tid == nthreads - 1 ? M : start + chunk);
        for (size_t i = start; i < end; ++i)
            local_sum += A[i] * B[i];
#pragma omp critical
        global_sum += local_sum;
    }
    return global_sum;
}

int main()
{
    const std::vector<size_t> Ms = {10000000, 50000000, 100000000};
    const int threads[] = {2, 4, 8};

    std::mt19937_64 rng(12345);
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    std::cout << std::fixed << std::setprecision(6);
    std::cout << "Method     |       M | Threads |   Time(s)\n";
    std::cout << "-----------------------------------------\n";

    for (size_t M : Ms)
    {
        std::vector<double> A(M), B(M);
        for (size_t i = 0; i < M; ++i)
        {
            A[i] = dist(rng);
            B[i] = dist(rng);
        }

        // serial
        double t0 = omp_get_wtime();
        double ds = dot_serial(A, B);
        double ts = omp_get_wtime() - t0;
        std::cout
            << std::setw(10) << "serial"
            << " | " << std::setw(8) << M
            << " | " << std::setw(7) << 1
            << " | " << std::setw(8) << ts
            << "\n";

        for (int t : threads)
        {
            omp_set_num_threads(t);

            // reduction
```

```
            t0 = omp_get_wtime();
            double dr = dot_reduction(A, B);
            double tr = omp_get_wtime() - t0;
            std::cout
                << std::setw(10) << "reduction"
                << " | " << std::setw(8) << M
                << " | " << std::setw(7) << t
                << " | " << std::setw(8) << tr
                << "\n";

            // atomic
            t0 = omp_get_wtime();
            double da = dot_atomic(A, B);
            double ta = omp_get_wtime() - t0;
            std::cout
                << std::setw(10) << "atomic"
                << " | " << std::setw(8) << M
                << " | " << std::setw(7) << t
                << " | " << std::setw(8) << ta
                << "\n";

            // manual
            t0 = omp_get_wtime();
            double dm = dot_manual(A, B);
            double tm = omp_get_wtime() - t0;
            std::cout
                << std::setw(10) << "manual"
                << " | " << std::setw(8) << M
                << " | " << std::setw(7) << t
                << " | " << std::setw(8) << tm
                << "\n";
        }
        std::cout << "-----------------------------------------\n";
    }

    return 0;
}
```

Таблицы с результатами экспериментов

```
./bin/pg5.1
Threads | reduction time (s) | atomic time (s)
1       | 0.000761092        | 0.0026282
1       | 0.00757379         | 0.026305
1       | 0.0760641          | 0.262022
-----------------------------------------
2       | 0.000785063        | 0.0099434
2       | 0.00377177         | 0.100218
2       | 0.037891           | 1.04466
-----------------------------------------
4       | 0.000480069        | 0.0118216
4       | 0.00166797         | 0.119219
4       | 0.017469           | 1.15275
-----------------------------------------
8       | 0.00050404         | 0.0119516
8       | 0.00151649         | 0.139889
8       | 0.0154056          | 1.33386
-----------------------------------------
```

```
./bin/pg5.2
Method     |       N | Threads |   Time(s) |    Error
-----------------------------------------------------------
    serial | 1000000 |       1 | 0.002760 | 0.000001
 reduction | 1000000 |       2 | 0.002065 | 0.000001
    atomic | 1000000 |       2 | 0.022321 | 0.000001
  critical | 1000000 |       2 | 0.036777 | 0.000001
 reduction | 1000000 |       4 | 0.001286 | 0.000001
    atomic | 1000000 |       4 | 0.028065 | 0.000001
  critical | 1000000 |       4 | 0.071802 | 0.000001
 reduction | 1000000 |       8 | 0.000882 | 0.000001
    atomic | 1000000 |       8 | 0.032547 | 0.000001
  critical | 1000000 |       8 | 0.113480 | 0.000001
-----------------------------------------------------------
    serial | 5000000 |       1 | 0.016366 | 0.000000
 reduction | 5000000 |       2 | 0.007987 | 0.000000
    atomic | 5000000 |       2 | 0.103223 | 0.000000
  critical | 5000000 |       2 | 0.182515 | 0.000000
 reduction | 5000000 |       4 | 0.004310 | 0.000000
    atomic | 5000000 |       4 | 0.127488 | 0.000000
  critical | 5000000 |       4 | 0.336428 | 0.000000
 reduction | 5000000 |       8 | 0.002279 | 0.000000
    atomic | 5000000 |       8 | 0.156786 | 0.000000
  critical | 5000000 |       8 | 0.563674 | 0.000000
-----------------------------------------------------------
    serial | 10000000 |       1 | 0.030814 | 0.000000
 reduction | 10000000 |       2 | 0.015569 | 0.000000
    atomic | 10000000 |       2 | 0.204792 | 0.000000
  critical | 10000000 |       2 | 0.342318 | 0.000000
 reduction | 10000000 |       4 | 0.007661 | 0.000000
    atomic | 10000000 |       4 | 0.269933 | 0.000000
  critical | 10000000 |       4 | 0.620423 | 0.000000
 reduction | 10000000 |       8 | 0.004224 | 0.000000
    atomic | 10000000 |       8 | 0.379833 | 0.000000
  critical | 10000000 |       8 | 1.123597 | 0.000000
-----------------------------------------------------------
```

```
./bin/pg5.3
Method     |        M | Threads |   Time(s)
-----------------------------------------------
    serial | 10000000 |       1 | 0.036372
 reduction | 10000000 |       2 | 0.019519
    atomic | 10000000 |       2 | 0.270992
    manual | 10000000 |       2 | 0.019110
 reduction | 10000000 |       4 | 0.012481
    atomic | 10000000 |       4 | 0.298865
    manual | 10000000 |       4 | 0.009630
 reduction | 10000000 |       8 | 0.009828
    atomic | 10000000 |       8 | 0.387534
    manual | 10000000 |       8 | 0.007160
-----------------------------------------------
    serial | 50000000 |       1 | 0.186254
 reduction | 50000000 |       2 | 0.098841
    atomic | 50000000 |       2 | 1.184447
    manual | 50000000 |       2 | 0.141307
 reduction | 50000000 |       4 | 0.090849
    atomic | 50000000 |       4 | 1.718968
    manual | 50000000 |       4 | 0.071969
 reduction | 50000000 |       8 | 0.048913
    atomic | 50000000 |       8 | 2.087962
    manual | 50000000 |       8 | 0.048511
-----------------------------------------------
    serial | 100000000 |       1 | 0.358412
 reduction | 100000000 |       2 | 0.196299
    atomic | 100000000 |       2 | 2.628974
    manual | 100000000 |       2 | 0.192449
 reduction | 100000000 |       4 | 0.146811
    atomic | 100000000 |       4 | 3.021313
    manual | 100000000 |       4 | 0.100716
 reduction | 100000000 |       8 | 0.088098
    atomic | 100000000 |       8 | 4.165211
    manual | 100000000 |       8 | 0.077104
-----------------------------------------------
```