1.

#include <omp.h>

#include <iostream>

int main(int argc, char \*argv[]) {

#pragma omp parallel for num\_threads(5)

{

/\* Данный код будет выполнен пятью потоками.

Части цикла будут поделены между

пятью потоками текущей группы потоков.\*/

for (int n = 1; n <= 10; ++n)

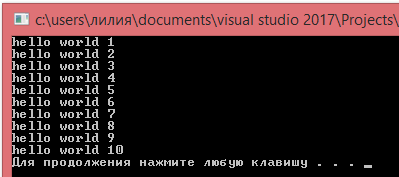
printf("hello world %d\n", n);

}

system("pause");

return 0;

}



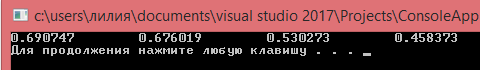
2.

Static - Ещё до входа в цикл каждый поток «знает», какие части цикла он будет обрабатывать.

Dymanic - В данном случае невозможно предсказать порядок, в котором итерации цикла будут назначены потокам. Каждый поток выполняет указанное число итераций. Если это число не задано, по умолчанию оно равно 1. После того как поток завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации. Последний набор итераций может быть меньше, чем изначально заданный. Такой вариант очень полезен, когда разные итерации цикла обсчитываются разное время.

Guided - Он похож на dynamic, но размер порции уменьшается экспоненциально.

Динамическое и управляемое планирование хорошо подходят, если при каждой итерации выполняются разные объемы работы или если одни процессоры более производительны, чем другие. При статическом планировании нет никакого способа, позволяющего сбалансировать нагрузку на разные потоки. При динамическом и управляемом планировании нагрузка распределяется автоматически - такова сама суть этих подходов. Как правило, при управляемом планировании код выполняется быстрее, чем при динамическом, вследствие меньших издержек на планирование.

1) Равномерное время выполнения

#include <omp.h>

#include <iostream>

int main()

{

int size = 100000000;

int \*a = new int[size];

omp\_set\_num\_threads(4);

int i;

double start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

for (i = 0; i < size; i++) a[i] = 1;

double stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

for (i = 0; i < size; i++) a[i] = 1;

stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided, 15)

for (i = 0; i < size; i++) a[i] = 1;

stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided, 4)

for (i = 0; i < size; i++) a[i] = 1;

stop = omp\_get\_wtime();

std::cout << stop - start << "\n";

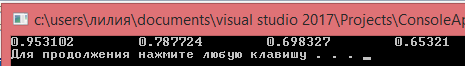
delete[] a;

system("pause");

return 0;

}

2) Нечетные итерации выполняются дольше



#include <omp.h>

#include <iostream>

int main()

{

int size = 100000000;

int \*a = new int[size];

omp\_set\_num\_threads(4);

int i;

double start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

for (i = 0; i < size; i++)

{

a[i] = 1;

if ((i % 2) != 0)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

double stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

for (i = 0; i < size; i++)

{

a[i] = 1;

if ((i % 2) != 0)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided, 15)

for (i = 0; i < size; i++)

{

a[i] = 1;

if ((i % 2) != 0)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided, 4)

for (i = 0; i < size; i++)

{

a[i] = 1;

if ((i % 2) != 0)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

stop = omp\_get\_wtime();

std::cout << stop - start << "\n";

delete[] a;

system("pause");

return 0;

}

3) Итерации в первой половине цикла выполняются дольше



#include <omp.h>

#include <iostream>

using namespace std;

int main()

{

int size1, size = 100000000;

size1 = size / 2;

int \*a = new int[size];

omp\_set\_num\_threads(4);

int i;

double start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

for (i = 0; i < size; i++)

{

a[i] = 1;

if (i <= size1)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

double stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

for (i = 0; i < size; i++)

{

a[i] = 1;

if (i <= size1)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided, 15)

for (i = 0; i < size; i++)

{

a[i] = 1;

if (i <= size1)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

stop = omp\_get\_wtime();

std::cout << stop - start << "\t";

start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided, 4)

for (i = 0; i < size; i++)

{

a[i] = 1;

if (i <= size1)

a[i] = a[i] \* a[i];

else a[i] = a[i] + 1;

}

stop = omp\_get\_wtime();

std::cout << stop - start << "\n";

delete[] a;

system("pause");

return 0;

}

3.

**Reduction:**

В этом случае каждый поток получит собственную копию переменной sum, выполнит с ней все необходимые действия, а затем произведет указанную операцию, чтобы объединить получившиеся значения с исходным.

#include <omp.h>

#include <iostream>

int main()

{

int size = 100000000;

int \*a = new int[size];

long int sum = 0;

omp\_set\_num\_threads(4);

int i;

double start = omp\_get\_wtime();

#pragma omp parallel for reduction (+: sum)

for (i = 0; i < size; i++) {

a[i] = 1;

sum += a[i];

}

double stop = omp\_get\_wtime();

std::cout << sum << "\n";

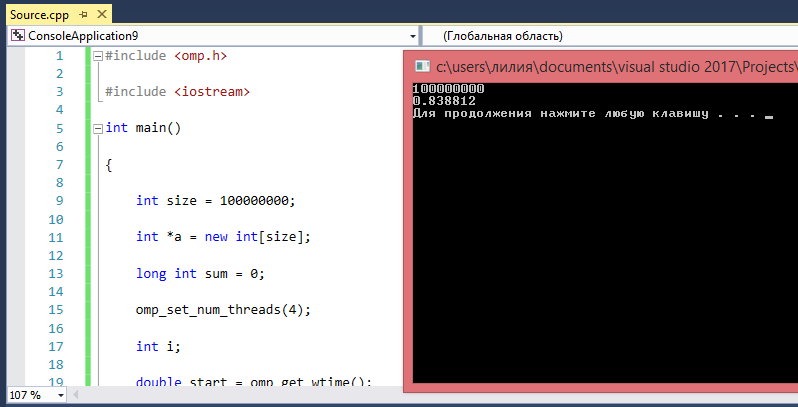
std::cout << stop - start << "\n";

delete[] a;

system("pause");

return 0;

}

****

**Critical:**

Если возникает необходимость выполнить какое-либо действие, изменяющее состояние такого объекта, из двух потоков, необходимо позаботиться о том, чтобы это действие выполнялось только одним из потоков в каждый момент времени. Для этого можно использовать директиву critical.

#include <omp.h>

#include <iostream>

int main()

{

int size = 100000000;

int \*a = new int[size];

long int sum = 0;

omp\_set\_num\_threads(4);

int i;

double start = omp\_get\_wtime();

#pragma omp parallel

for (i = 0; i < size; i++)

{

a[i] = 1;

#pragma omp critical

sum += a[i];

}

double stop = omp\_get\_wtime();

std::cout << sum << "\n";

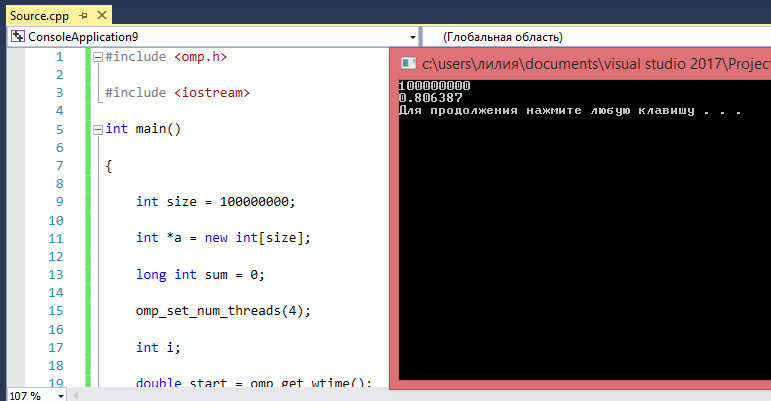
std::cout << stop - start << "\n";

delete[] a;

system("pause");

return 0;

}



4.

Барьеры устанавливаются в тех участках кода, в которых необходимо синхронизировать все потоки. То есть по достижении барьера, поток будет ждать пока другие потоки также достигнут этого барьера, после чего выполнение работы продолжится.

Директива barrier используется для упорядочивания вывода от работающих нитей. Выдачи с разных нитей "1" и "2" могут перемежаться в

произвольном порядке, а выдача "3" со всех нитей придёт строго

после двух предыдущих выдач.

#include <stdio.h>

#include <omp.h>

#include <iostream>

using namespace std;

int main()

{

#pragma omp parallel

{

cout << 1 << endl;

cout << 2 << endl;

#pragma omp barrier

cout << 3 << endl;

}

system("pause");

}

5.

#include <omp.h>

#include <stdio.h>

#include <iostream>

#include <cstdlib>

using namespace std;

int main() {

int n = 5;

int g = 1;

int \*\*A = new int\*[n];

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

A[i] = new int[n + 1];

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

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

A[i][j] = 1;

cout << A[i][j] << " ";

g++;

}

cout << endl;

}

cout << endl;

omp\_set\_num\_threads(4);

#pragma omp parallel num\_threads(4)

{

int r, c;

if (omp\_get\_thread\_num() == 0)

r = 0, c = 0;

if (omp\_get\_thread\_num() == 1)

r = 1, c = 0;

if (omp\_get\_thread\_num() == 2)

r = 0, c = 1;

if (omp\_get\_thread\_num() == 3)

r = 1, c = 1;

for (int i = 2 + c; i < n; i += 2)

{

for (int j = 2 + r; j < n; j += 2)

{

A[i][j] = A[i - 2][j] + A[i][j - 2];

}

}

}

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

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

cout << A[i][j] << " ";

}

cout << endl;

}

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

A[i] = new int[n + 1];

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

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

A[i][j] = 1;

g++;

}

}

cout << endl;

for (int i = 2; i < n; i++)

{

for (int j = 2; j < n; j++)

{

A[i][j] = A[i - 2][j] + A[i][j - 2];

}

}

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

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

cout << A[i][j] << " ";

}

cout << endl;

}

system("PAUSE");

}