Санкт-Петербургский
политехнический университет Петра Великого

Лабораторная работа №2:

*«Изучение базовых конструкций технологии OpenMP
на примере параллезации программы вычисления определенного интеграла»*

Группа: 3640301/00201
Студент: Машаев Н.К.
Преподаватель: Абрамов А.Г.

Санкт-Петербург 2020

Содержание

1 О работе

1.1 Цель работы

Целью работы являются изучение базовых принципов работы с общими и локальными переменными, освоение применения синхронизирующих конструкций технологий OpenMP.

1.2 Задание к работе

Разработать программу, вычисляющую определенный интеграл (1) на конечном промежутке методом прямоугольников, и выполнить ее распараллеливание средствами OpenMP.

$$\int_0^1 \frac{\ln x}{1-x} dx = -\frac{\pi^2}{6} \approx -1.64493406685 \tag{1}$$

1.3 Ход выполнения работы

1. Разработать и отладить последовательную версию программы. Провести проверку правильности ее выполнения, сопоставив с "табличным" значением интеграла;

2. Провести исследование возможностей и эффективности автоматического распараллеливания программ компилятором g++, проверить влияние опций оптимизации на время выполнения программ;

3. Выполнить распараллеливание последовательной версии программы следующими способами:

   - SMPD - вариант, предполагающий использование только директивы omp parallel и явное распределение между нитями итераций реализующего процесс интегрирования цикла (при этом идентификатор нити используется в качетсве стартового индекса цикла для данной нити, а количество нитей - в качестве шага цикла);
   - вариант с применением OpenMP-директивы автоматической параллелизации цикла omp for и сохранением результатов частичного суммирования для каждой из нитей в общем массиве (размерность массива равна количеству нитей);
   - вариант с применением OpenMP-директивы автоматической параллелизации цикла omp for, определение локальной вещественной переменной для хранения результатов частичного суммирования в каждой из нитей с использованием директивы CRITICAL для сложения рассчитанных нитями значений частичных сумм;
   - вариант с примененим OpenMP-директивы автоматической параллелизации цикла omp for, определение локальной вещественной переменной для хранения результатов частичного суммирования в каждой из нитей

и с использованием директивы ATOMIC для сложения расчитанных нитями значений частичных сумм;

- вариант с применением OpenMP-директивы автоматической параллелизации цикла omp for, определением локальной вещственной переменной для хранения результатов частичного суммирования в каждой из нитей с использованием механизма синхронизации на основе замков (семафоров) для сложения итоговых значений частичных сумм;
- вариант с применением OpenMP-директивы автоматической параллелизации цикла DO, определением локальной вещественной переменной для хранения результатов частичного суммирования в каждой из нитей и использованием операции редукции(REDUCTION) для сложения расчитанных нитями значений частичных сумм.

4. Провести исследования эффективности распараллеливания программы для реализованных способов, задавая разное число нитей. Оптимизация при компиляции должна быть отключена (ключ -O0);

5. Для последнего варианта программы (с операцией REDUCTION) провести исследование влияния на эффективность параллелизации способа распределения итераций цикла между нитями (STATIC, DYNAMIC, GUIDED), а также размера блока распределяемых итераций для статического способа (исследование проводить для фиксированного числа работающих нитей, например, для четырех).

2 Исследование возможностей OpenMP

Программа написана на языке программирования C++. Для работы с функциями реализован модуль functions.h(functions.cpp). Здесь определен абстрактный класс IFunction, имеющий чисто виртуальный метод get_value(x), который вычисляет значение функции в точке x. Конкретная реализация данного метода производится в классе-наследнике Function_6, который по своей сути отвечает за работу с подынтегральной функции в формуле (1). Промежуток интегрирования разбивался на 100000000 частей.Время выполнения вычислений измерялось при помощи стандартного модуля <chrono>.

2.1 Отладка последовательной версии программы

Последовательная версия программы для вычисления определенного интеграла (1) представлена ниже:

Листинг 2.1: *Последовательная версия программы вычисления определенного интеграла (consistent_integral)*

```
1  double sum = 0.0;
2  double curr_x = 0.0;
3  for (unsigned int i = 0; i < chunk_numb; ++i)
4  {
```

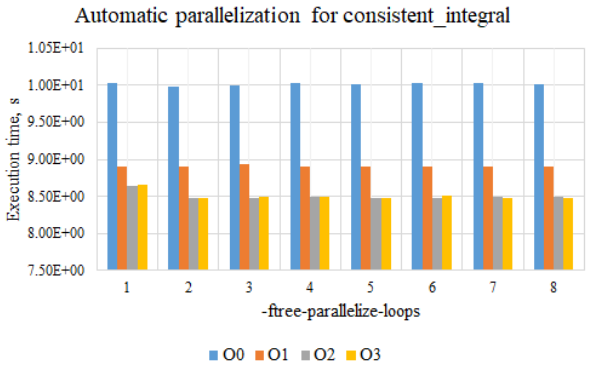
```
5       curr_x = x_start + (i + 0.5) * x_step;
6       sum += func.get_value(curr_x);
7   }
8   double integration_result = sum * x_step;
```

Полученный в результате выполнения программы, приведенной выше, составляет -1.64493406685. Абсолютная разность с точным решением (1) составляет $3.46589046352 \cdot 10^{-9} \approx 3.47 \cdot 10^{-9}$. Таким образом, погрешность относительно точного решения менее одного процента, что свидетельствует о правильности вычислений приведенной программы.

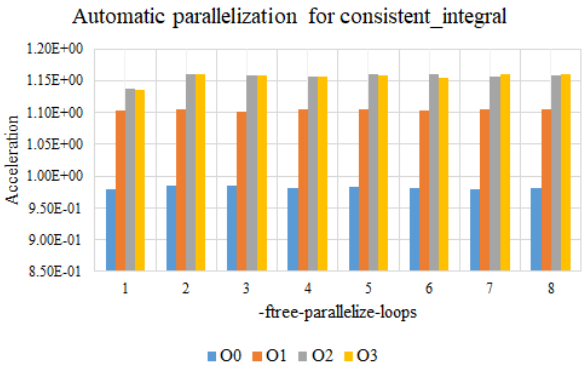
2.2 Автоматическое распараллеливание

Компиляторы gnu позволяют выполнять автоматическое распараллеливание последовательного кода.

Для этого необходимо использовать ключ $-ftree - parallelize - loops = n$, который распараллеливает циклы, разделяя пространство итераций между n нитями. Стоит отметить, что параллелизация возможна лишь для циклов, у которых итерации не зависят друг от друга и от порядка их следования в цикле.



(a) Зависимость времени выполнения от числа нитей



(b) Зависимость ускорения от числа нитей

Рис. 1: Исследование возможностей автоматического распараллеливания для программы вычисления определенного интеграла

Параметры		consistent_integral	
Оптимизация	**ftree-parallelize-loops**	**Time**	**Acceleration**
O0	1	10.0325	0.9800
	2	9.9881	0.9844
	3	9.9890	0.9843
	4	10.0242	0.9808
	5	10.0051	0.9827
	6	10.0232	0.9809
	7	10.0364	0.9796
	8	10.0151	0.9817
O1	1	8.9062	1.1040
	2	8.8969	1.1051
	3	8.9287	1.1012
	4	8.9035	1.1043
	5	8.9010	1.1046
	6	8.9066	1.1039
	7	8.9010	1.1046
	8	8.9013	1.1046
O2	1	8.6490	1.1368
	2	8.4789	1.1596
	3	8.4867	1.1585
	4	8.4952	1.1574
	5	8.4786	1.1596
	6	8.4789	1.1596
	7	8.4953	1.1574
	8	8.4927	1.1577
O3	1	8.6522	1.1364
	2	8.4782	1.1597
	3	8.4909	1.1579
	4	8.4953	1.1574
	5	8.4864	1.1586
	6	8.5116	1.1551
	7	8.4790	1.1596
	8	8.4786	1.1596

Таблица 1: Исследование возможностей автоматического распараллеливания для программы вычисления определенного интеграла consistent_integral

Стоит отметить что ускорение в таблице 1 расчитывалось как отношение времени выполнения программы без ключа $-ftree-parallelize-loops$ (для программы вычисления определенного интеграла consistent_integral составляет 9.8321 с.) к времени выполнения с ключом с заданным количеством нитей.

Из таблицы 1 и рис.1 видно, что применение автоматического распараллеливания не влечет за собой увеличения производительности программы.

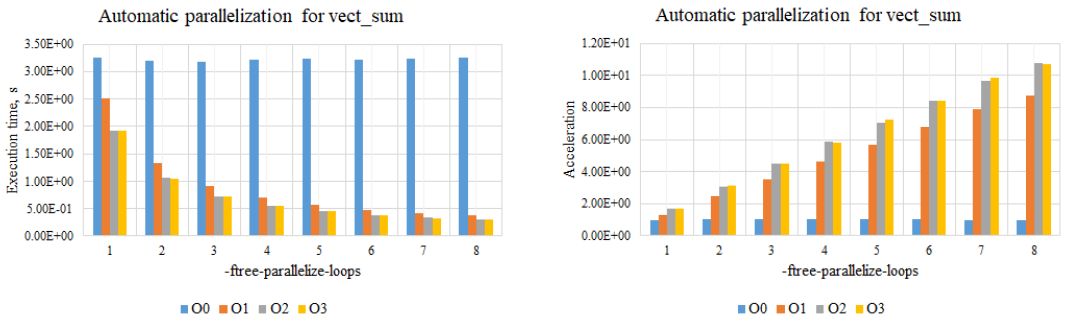
При вычислении определенного интеграла отсутствует работа с массивами. Интересным также представляется исследования влияния ключа $-ftree-parallelize-loops$ при работе с программами, оперирующих с векторными операциями. Для

изучения данного вопроса была реализована программа инициализации и вычисления суммы векторов, код которой приведен ниже:

Листинг 2.2: *Последовательная версия программы инициализации векторов и вычисления их суммы (vect_sum)*

```
for (unsigned int i = 0; i < vect_size; ++i)
{
  if (i % 2 == 0)
  {
    vect1[i] = 0.0;
    vect2[i] = 1.0;
  }
  else
  {
    vect1[i] = 1.0;
    vect2[i] = 0.0;
  }

  vects[i] = vect1[i] + vect2[i];
}
```

Здесь инициализируются два вектора размером 100000000 элементов. Для первого вектора все компоненты с четными индексами равны нулю, с нечетным - единице. Для второго - наоборот. Соответственно, их сумма vects - единичный вектор размером 100000000.



(a) Зависимость времени выполнения от числа нитей

(b) Зависимость ускорения от числа нитей

Рис. 2: Исследование возможностей автоматического распараллеливания для программы инициализации и вычисления суммы векторов

Ускорение в таблице 2 рассчитывалось аналогично случаю вычисления интеграла, т.е. как отношение времени выполнения программы без ключа $-ftree-parallelize-loops$ (для программы вычисления определенного интеграла vect_sum составляет 3.2634 с.) к времени выполнения с ключом с заданным числом нитей.

Согласно данным, приведенным в таблице 2 и на рис.2 в случае отстутствия оптимизации ключ $-ftree-parallelize-loop$ не влияет на скорость выполнения

программы, однако применение данной опции при наличии оптимизации влечет к ускорению выполнения кода.

Параметры		**consistent_integral**	
Оптимизация	**ftree-parallelize-loops**	**Time**	**Acceleration**
O0	1	3.2557	1.0024
	2	3.1957	1.0212
	3	3.1784	1.0268
	4	3.2089	1.0170
	5	3.2281	1.0109
	6	3.2132	1.0156
	7	3.2300	1.0104
	8	3.2537	1.0030
O1	1	2.5153	1.2975
	2	1.3333	2.4476
	3	0.9181	3.5547
	4	0.7055	4.6256
	5	0.5758	5.6674
	6	0.4818	6.7728
	7	0.4144	7.8745
	8	0.3733	8.7410
O2	1	1.9312	1.6898
	2	1.0566	3.0885
	3	0.7284	4.4805
	4	0.5582	5.8464
	5	0.4624	7.0580
	6	0.3875	8.4212
	7	0.3374	9.6730
	8	0.3028	10.7772
O3	1	1.9283	1.6924
	2	1.0397	3.1390
	3	0.7299	4.4713
	4	0.5610	5.8167
	5	0.4521	7.2179
	6	0.3866	8.4405
	7	0.3309	9.8612
	8	0.3058	10.6716

Таблица 2: Исследование возможностей автоматического распараллеливания для программы инициализации и суммирования векторов vect_sum

Таким образом, использование ключа $-ftree-parallelize-loops$ существенно сказывается на эффективности выполнения программы с массивами, где возникает вопрос работы с кеш памятью, при наличии оптимизации. В случае отсутствие оптимизации или компиляции кода программ, где работа с массивами отсутствует, применение автоматической параллелизации не дает прироста производительности.

2.3 Параллелизация программы методом SMPD

Параллелизация программы методом SMPD предполагает, что внутри параллельной области распределение работы между нитями производится вручную, явным определением для каждой из нитей обрабатываемых в цикле элементов векторов. При назначении блока итераций цикла для конкретной нити используются ее номер (идентификатор) и общее количество формирующих группу нитей. В рамках широко известной классификации Флинна можно считать, что данный метод реализует парадигму параллелизма **S**ingle **P**rocess **M**ultiply **D**ata. Отсюда и название данного метода параллелизации SMPD.

Листинг 2.3: Параллелизация программы методом SMPD

```
#pragma omp parallel shared(x_start, x_step) \
private(curr_x, func)    \
reduction(+:sum)
{
  unsigned int nthreads = omp_get_num_threads();
  unsigned int curr_tid = omp_get_thread_num();
  for (unsigned int i = curr_tid;
  i < chunk_numb;
  i += nthreads)
  {
    curr_x = x_start + (i + 0.5) * x_step;
    sum += func.get_value(curr_x);
  }
}
double integration_result = sum * x_step;
```

При реализации параллелизации методом SMPD в данной лабораторной работе результаты частичного суммирования для каждой из нитей сохраняются в локальной переменной при помощи атрибута reduction.

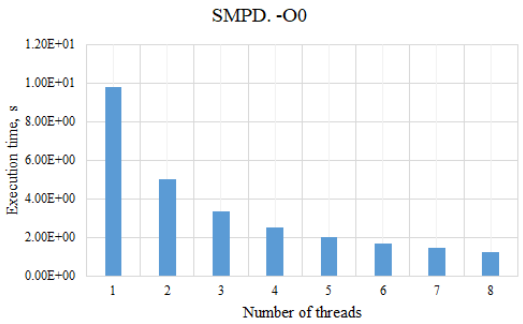
Изучения эффективности распараллеливания будем производить увеличивая число нитей от 1 до 8. При этом исключим влияние оптимизации, установив ключ $-O0$.

Число нитей	Время,с	Ускорение
1	9.8078	1.0025
2	4.9934	1.9690
3	3.3312	2.9515
4	2.4996	3.9335
5	2.0003	4.9153
6	1.6668	5.8986
7	1.4283	6.8838
8	1.2499	7.8660

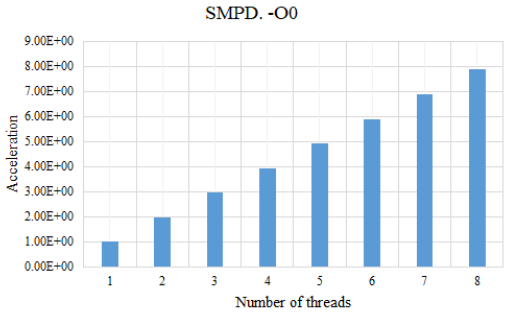
Таблица 3: SMPD параллелизация

Здесь и далее будем считать за ускорение отношение времени выполнения последовательной версии программы (для программы вычисления определенного

интеграла consistent_integral составляет 9.8321 с.) к времени выполнения параллельной.



(a) Зависимость времени выполнения от числа нитей



(b) Зависимость ускорения от числа нитей

Рис. 3: Исследование применения SMPD метода параллелизации

Согласно данным, приведенным в таблице 3 и на рис.3 при увеличении числа нитей время выполнения программы стабильно уменьшается. Стоит отметить, что подобная "ручная" настройка выполнения цилов гарантирует равномерное распределение работы между нитями, что и одной из причин роста производительности с увеличением числа нитей.

2.4 Параллелизация с сохранением результатов частичного суммирования в массиве

Выполним параллелизацию вычисления определенного интеграла (1) при помощи директивы omp for.

При использовании параллелизации в данном случае возникает проблема с сохранением результатов суммирования, полученного каждой из нитей, в общей переменной. Если сделать данную переменную общей для всех нитей, то нити будут хаотичным образом обращаться к ней, что приведет к некорректным результатам.

Для решения возникшей проблемы результаты частичного суммирования в данном пункте лабораторной работы хранятся в массиве размерностью, равной числу нитей. Каждая нить имеет доступ лишь к одному собственному элементу массива, в результате чего обеспечивается независимость хранения данных в параллельной области между нитями, но при этом доступ к элементам массива имеется извне параллельной директивы, где может быть произведена дальнейшая обработка полученных каждой нитей результатов.

Листинг 2.4: Параллелизация с сохранением результатов частичного суммирования в массиве (for_tid):

```
#pragma omp parallel shared(x_start, x_step, \
nthreads,          \
chunk_numb,        \
thread_sum)        \
private(curr_x, func)
```

```
6   {
7     #pragma omp master
8     {
9       nthreads = omp_get_num_threads();
10      thread_sum = new double[nthreads];
11      for (unsigned int i = 0; i < nthreads; ++i)
12      {
13        thread_sum[i] = 0.0;
14      }
15    }
16
17    #pragma omp barrier
18
19    unsigned int curr_tid = omp_get_thread_num();
20    #pragma omp for schedule(runtime)
21    for (unsigned int i = 0; i < chunk_numb; ++i)
22    {
23      curr_x = x_start + (i + 0.5) * x_step;
24      thread_sum[curr_tid] += func.get_value(curr_x);
25    }
26  }
27
28  double integration_result = 0.0;
29  for (unsigned int i = 0; i < nthreads; ++i)
30  {
31    integration_result += thread_sum[i];
32  }
33  integration_result *= x_step;
```

Поскольку число нитей задается при помощи переменной окружения, то для выделения памяти массива частичного суммирования используется директива omp master.

Директива omp master гарантирует, что код внутри нее будет выполнен одной мастер нитью (нитью с идентификатором 0). Здесь происходит вычисление текущего числа нити, аллокация памяти под массив и его инициализация.

Поскольку остальные нити не ждут выполнения директивы omp master, то для того чтобы предотвратить обращению нити к еще не выделенной области памяти следует поставить баррьер omp barrier, который гарантирует то, что дальнейший кусок кода не будет выполняться, пока все нити не достигнут данного баррьера.

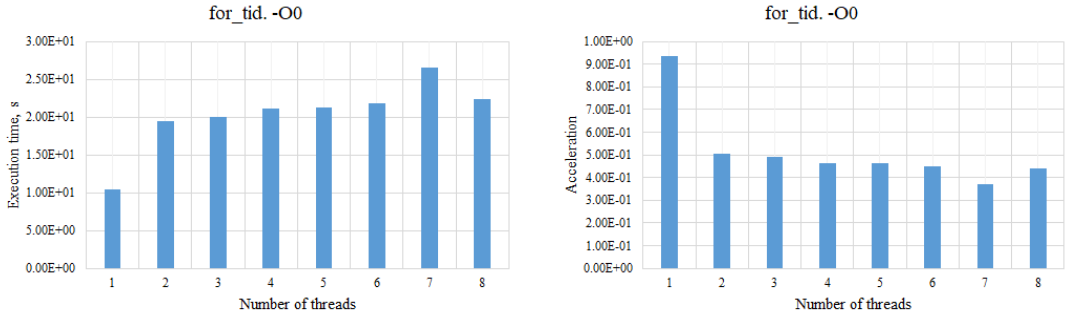
Ниже на рис.4, построенном по данным таблицы 4, приведены результаты исследования эффективности параллелизации с сохранением результатов частичного суммирования в массиве.

Как видно из рис.4 данный вариант параллелизации не является эффективным для решения задачи вычисления определенного интеграла. Причина ухудшения производительности относительно последовательной версии и хаотичного изменения скорости выполнения программы с ростом числа нитей является то, что в распараллеленной области порядок обращения нитями к массиву произвольный. В результате в параллельной области происходит одновременное обращение ни-

тями к элементам массива в хаотичном порядке, что ведет к неэффективному использованию кеш памяти.

Число нитей	Время,с	Ускорение
1	10.4874	0.9375
2	19.4019	0.5068
3	19.9867	0.4919
4	21.1888	0.4640
5	21.2392	0.4629
6	21.7672	0.4517
7	26.5952	0.3697
8	22.4161	0.4386

Таблица 4: Исследование параллелизации с сохранением результатов частичного суммирования в массиве



(a) Зависимость времени выполнения от числа нитей

(b) Зависимость ускорения от числа нитей

Рис. 4: Исследование параллелизации с сохранением результатов частичного суммирования в массиве

2.5 Использование директивы critical

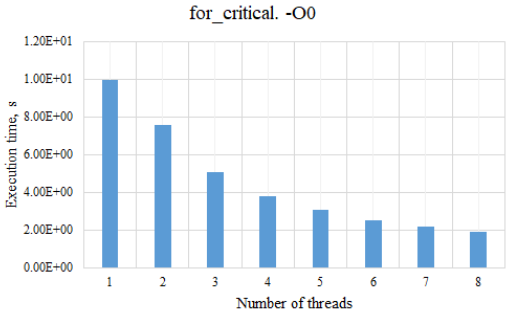
Хранение результатов частичного суммирования при помощи массива в параллельной области памяти оказалось неэффективным. Для записи данных суммирования воспользуемся директивой omp critical.

Директива critical реализует в OpenMP синхронизирующую операцию взаимного исключения. Это означает, что в каждый момент времени в связанном с критической секцией структурированном блоке может находиться только одна нить. Другие нити, достигнув начала критической секции, будут ожидать завершения выполнения этой нитью кода программы, соответствующего структурированному блоку, после чего одна из ожидающих нитей, выбранная случайным образом, "займет" критическую секцию.

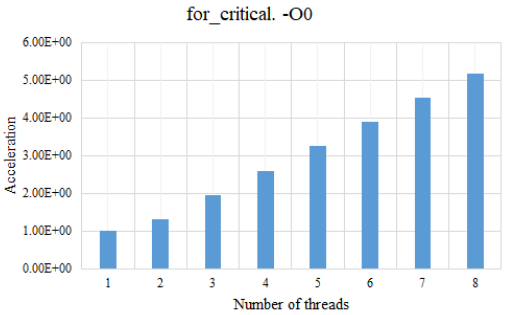
В данном случае критическая секция применяется для обеспечения сложения вычисленных частичных сумм в общую переменную *sum* и получения окончательного результата интегрирования

Листинг 2.5: Использование директивы CRITICAL(for_critical)

```
double curr_x = 0.0;
double sum = 0.0;
#pragma omp parallel shared(x_start, x_step, \
sum,                \
chunk_numb)         \
private(curr_x, func)
{
  double thread_sum = 0.0;

  #pragma omp for schedule(runtime)
  for (unsigned int i = 0; i < chunk_numb; ++i)
  {
    curr_x = x_start + (i + 0.5) * x_step;
    thread_sum += func.get_value(curr_x);
  }

  #pragma omp critical
  {
    sum += thread_sum;
  }
}

double integration_result = sum * x_step;
```



(a) Зависимость времени выполнения от числа нитей



(b) Зависимость ускорения от числа нитей

Рис. 5: Исследование применения omp critical

Число нитей	Время,с	Ускорение
1	9.9280	0.9903
2	7.5731	1.2983
3	5.0561	1.9446
4	3.7906	2.5938
5	3.0321	3.2427
6	2.5258	3.8926
7	2.1674	4.5363
8	1.8967	5.1838

Таблица 5: Исследование применения omp critical

Проведенные исследования (рис.5, таблица 5) показывают, что с увеличением числа нитей время выполнения программы убывает, а значит эффективность вычислений растет. В отличие от способа параллелизации с сохранением частичных сумм в массиве здесь не происходит дорогостоящие операция выделения памяти под массив и синхронизация директивой barrier. Также отсутствует многократное обращение к общим данным в хаотичном порядке, из-за чего с увеличением нитей время работы программы стабильно падает,а ускорение относительно последовательной версии растет.

2.6 Применение директивы atomic

Директиву omp critical можно заменить менее дорогостоящей omp atomic.

Директива atomic обеспечивает атомарную работу с заданной областью оперативной памяти, то есть гарантируется, что следующая непосредственно за директивой операция над этой областью будет выполняться непрерывно (неделимо), как единое целое. Кроме того, на время выполнения атомарной операции к этой области памяти будет заблокирован доступ на чтение и запись со стороны других нитей.

Директиву atomic можно рассматривать как упрощенную и менее дорогостоящую версию директивы critical, применяемую для синхронизации выполнения относительно простых действий над общими переменными. Вместе с тем, поскольку многие из допустимых для директивы atomic операций на аппаратном уровне выполняются как атомарные, ее использование в таких случаях является предпочтительным способом организации взаимного исключения и оказывается весьма эффективным на практике.

Директива atomic применяется для атомарного выполнения операции сложения вычисленных разными нитями частичных сумм.

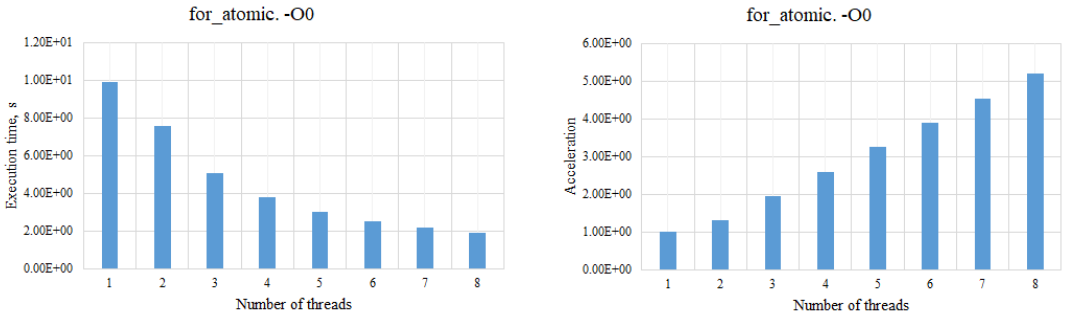
Листинг 2.6: Использование директивы atomic(for_atomic)

```
1  double curr_x = 0.0;
2  double sum = 0.0;
3  #pragma omp parallel shared(x_start, x_step, \
4  sum,                \
5  chunk_numb)         \
6  private(curr_x, func)
7  {
```

```
8       double thread_sum = 0.0;
9
10      #pragma omp for schedule(runtime)
11      for (unsigned int i = 0; i < chunk_numb; ++i)
12      {
13        curr_x = x_start + (i + 0.5) * x_step;
14        thread_sum += func.get_value(curr_x);
15      }
16
17      #pragma omp atomic
18      sum += thread_sum;
19    }
20
21    double integration_result = sum * x_step;
```

Число нитей	Время,с	Ускорение
1	9.8968	0.9935
2	7.5623	1.3001
3	5.0492	1.9473
4	3.7867	2.5965
5	3.0279	3.2472
6	2.5257	3.8928
7	2.1653	4.5407
8	1.8912	5.1988

Таблица 6: Исследование применения omp atomic



(a) Зависимость времени выполнения от числа нитей

(b) Зависимость ускорения от числа нитей

Рис. 6: Исследование применения omp atomic

Согласно рис.6 с увеличением числа нитей скорость вычисления интеграла растет. Производительность программы с использованием директивы omp atomic близка к производительности программы с использованием omp sections(рис.5). Сравнение времени выполнения программ приведено на рис.7.

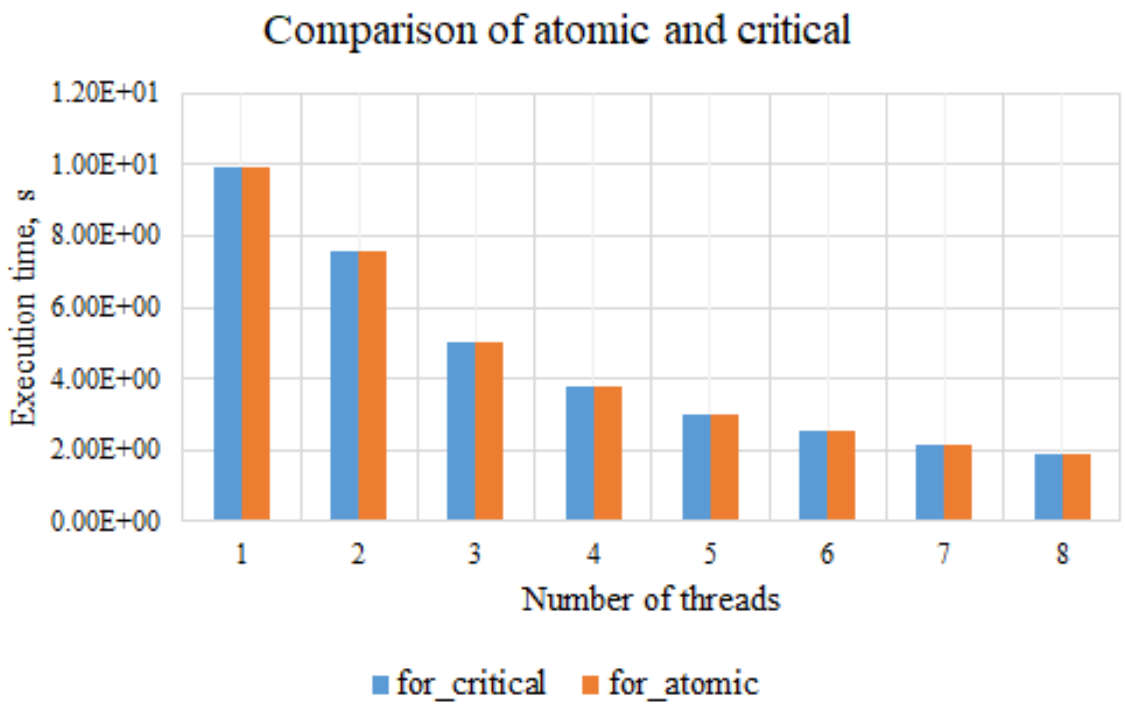


Рис. 7: Сравнение omp critical и omp atomic

2.7 Параллелизация с использованием механизма синхронизации на основе замков

В общем случае семафор (переменная-замок) представляет собой переменную-счетчик, которая может иметь любое целое неотрицательное значение и допускает применение к себе элементарных операций по инициализации, захвату и освобождению. Семафоры могут быть использованы для запрета одновременного выполнения несколькими нитями заданных участков кода программы. С установленной переменной-семафором в программе ассоциируется очередь ожидающих нитей. Нити, захватившей замок, разрешается выполнение защищаемого им участка кода программы.

В **OpenMP** замки представляют собой общие переменные целого типа, которые могут использоваться в программе только в качестве параметров соответствующего набора процедур синхронизации. **OpenMP**-замок может находиться в одном из трех состояний: неинициализированный, открытый или закрытый. В том случае, если замок находится в разблокированном состоянии, нить (выполняемая ею задача) может захватить замок, при этом его состояние изменится на "закрытый". Нить на время становится владельцем данного замка, и впоследствии может снять блокировку, возвратив его в состояние "открытый".

В данной задаче внутри параллельной области с помощью процедур захвата и освобождения переменной-замка реализуется защищенный, последовательный доступ нитей к операции обновления общей переменной хранения частичной суммы.

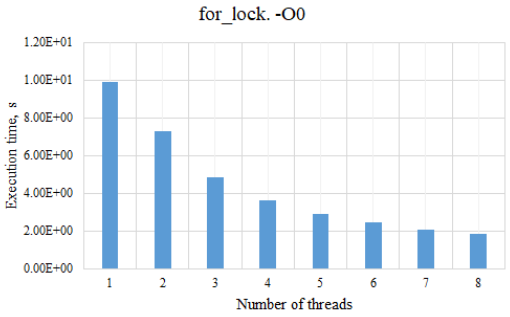
Листинг 2.7: Параллелизация с использованием механизма синхронизации на основе замков для сложения итоговых значений частичных сумм (for_lock):

```
  double curr_x = 0.0;
  double sum = 0.0;
  omp_lock_t lock;
  omp_init_lock(&lock);
  #pragma omp parallel shared(x_start, x_step, \
  sum,                \
  chunk_numb)         \
  private(curr_x, func)
  {
    double thread_sum = 0.0;

    #pragma omp for schedule(runtime)
    for (unsigned int i = 0; i < chunk_numb; ++i)
    {
      curr_x = x_start + (i + 0.5) * x_step;
      thread_sum += func.get_value(curr_x);
    }

    omp_set_lock(&lock);

    sum += thread_sum;

    omp_unset_lock(&lock);
  }
  omp_destroy_lock(&lock);
  double integration_result = sum * x_step;
```

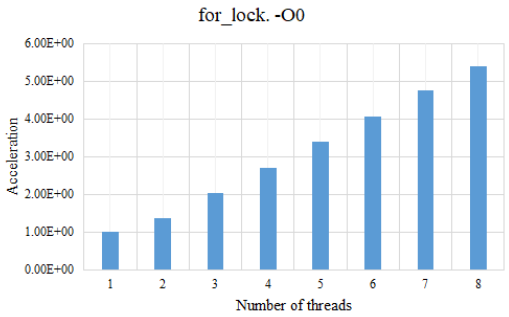
Применение omp lock в сочетании с приватными переменными предотвращает многократное хаотическое обращение нитей к общей области памяти. За равномерное распределение итераций между нитями отвечает директива omp for. В результате применение семафора влечет к уменьшению времени выполнения программы с увеличением числа нитей (рис.8, таблица 7).

Число нитей	**Время,с**	**Ускорение**
1	9.9052	0.9926
2	7.2641	1.3535
3	4.8423	2.0305
4	3.6364	2.7038
5	2.9078	3.3813
6	2.4253	4.0540
7	2.0741	4.7405
8	1.8190	5.4051

Таблица 7: Исследование применения omp lock



(a) Зависимость времени выполнения от числа нитей



(b) Зависимость ускорения от числа нитей

Рис. 8: Исследование применения omp lock

2.8 Автоматическая параллелизация цикла с использованием редукции

Специальный тип переменных, задаваемый атрибутом-оператором reduction, объединяет в себе черты общих и локальных переменных OpenMP и применяется к переменным, с которыми в параллельной области необходимо выполнить некоторую операцию совместной обработки (операция привидения или редукции).

Переменную, заданную при помощи reduction, можно использовать для хранения частичных сумм, получаемых нитей, с последующем суммированием результатов. Принцип работы переменной типа reduction следующий: в параллельной области обращения к переменной такого типа заменяются на обращения к ее копиям для каждой из нитей, а по завершению параллельной области заданная операция (или процедура) выполняется над окончательными значениями локальных копий переменной во всех нитях. Ее результат присваивается оригинальной переменной и сохраняется в главной нити.

Листинг 2.8: Автоматическая параллелизация цикла с использованием редукции(for_reduction):

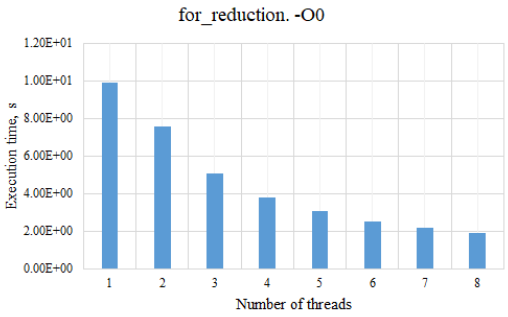
```
1  double curr_x = 0.0;
2  double sum = 0.0;
3  #pragma omp parallel shared(x_start, x_step, \
4  chunk_numb)       \
5  private(curr_x, func)    \
6  reduction(+: sum)
7  {
8    #pragma omp for schedule(runtime)
9    for (unsigned int i = 0; i < chunk_numb; ++i)
10   {
11     curr_x = x_start + (i + 0.5) * x_step;
12     sum += func.get_value(curr_x);
13   }
14 }
```

```
15
16    double integration_result = sum * x_step;
```

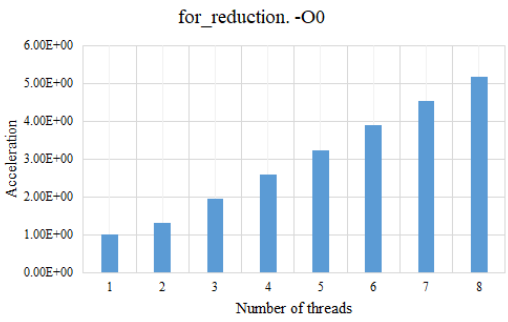
Директива omp_for в данном случае обеспечивает равномерную нагрузку нитей и, в сочетании с атрибутом reduction, предотвращает многократное хаотичное обращение нитей к общим переменным, что ведет к росту производительности с увеличением числа нитей.

Число нитей	Время,с	Ускорение
1	9.8678	0.9964
2	7.5824	1.2967
3	5.0593	1.9434
4	3.7968	2.5895
5	3.0357	3.2388
6	2.5299	3.8863
7	2.1687	4.5337
8	1.8992	5.1770

Таблица 8: Исследование применения reduction



(a) Зависимость времени выполнения от числа нитей



(b) Зависимость ускорения от числа нитей

Рис. 9: Исследование применения reduction

Сравнение эффективности реализованных подходов параллельного вычисления определенного интеграла относительно последовательной версии приведена на рис.10. Для тех случаев, где применялась директива omp for в качестве переменной OMP_SCHEDULE использовалось значение STATIC,1. Из приведенных данных видно, что наихудшие показатели эффективности показывает метод параллелизации с хранением результатов частичного суммирования в массиве. Наилучшую производительность удалось достичь при распределении нагрузки между нитями "вручную" (метод SMPD), основным достоинством которого заключается в более тонком контроле нагрузки между нитями. Остальные методы также показывают хорошие результаты, достигая ускорения в 5 раз при восьми нитях относительно последовательной версии.

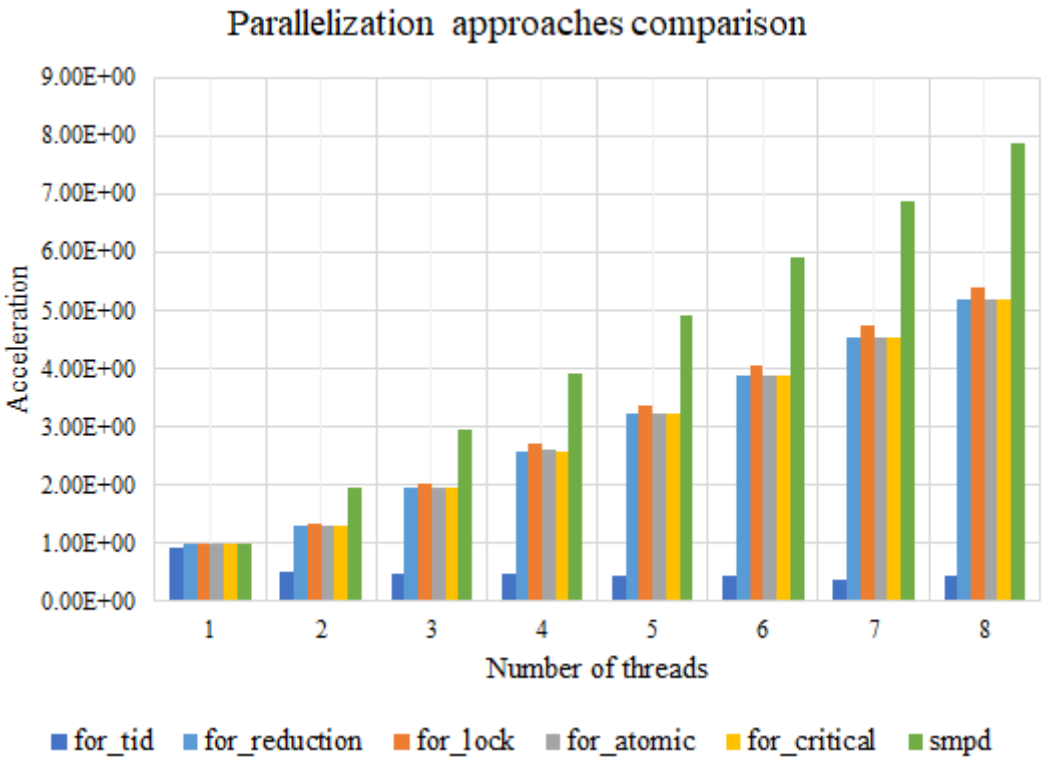


Рис. 10: Сравнение реализованных способов параллелизации

2.9 Исследования влияния на эффективность параллелизации способа распределения итераций цикла между нитями

Хотя метод SMPD дает более тонкую настройку распределения нагрузки между нитями, результаты для применения директивы omp for можно попытаться значительно улучшить подобрав оптимальный параметр атрибута schedule. В OpenMP определены следующие настройки атрибута schedule:

- Статический способ STATIC позволяет организовать контролируемое, заранее, до выполнения цикла, известное для конкретной реализации распределение итераций цикла между нитями. В отсутствие заданного параметра CHUNK_SIZE весь набор индексов цикла будет разделен на непрерывные порции примерно одинакового размера (согласованно с количеством нитей), с последующим упорядоченным распределением полученных порций между нитями в порядке роста их номеров (размер порции при этом не специфирован и зависит от реализации). Заданное значение CHUNK_SIZE устанавливает размер блока итераций при блочно-циклическом способе их распределения, таком, что нулевая нить выполняет первый по порядку роста индексов блок итераций, первая нить - следующий за ним блок и т.д. Если на последней по номеру нити выполнение цикла еще не завершилось, распределение оставшихся блоков итераций начнется снова, с нулевой нити, и так далее, в циклическом порядке, пока не закончатся все блоки.

- Динамический способ (DYNAMIC) также подразумевает разделение итераций цикла на блоки, размер которых отвечает заданному значению параметра CHUNK_SIZE, однако, в отличие от статического способа, назначение блоков на выполнение конкретным нитям производится не циклически, а с учетом из текущей занятости. Следующий по очереди блок итераций отдается на выполнение той нити, которая первой завершит выполнение операций в пределах своего, назначенного ей ранее блока. Таким образом, распределение итераций между нитями при динамическом способе, в общем случае, является непредсказуемым. Этот может не позволить эффективно использовать преимущества кэш-памяти по причине нарушения локальности данных. Вместе с тем, такой способ, в сравнении со статическим, является более гибким и в отдельных случаях улучшает балансировку загрузки нитей. Следует иметь в виду, что для динамического способа накладные расходы на выполнение планирования могут заметно превосходить имеющие место при других вариантах распределения. Улучшить ситуацию может подбор оптимального для каждой конкретной задачи значения параметра CHUNK_SIZE. По умолчанию этот параметр считается равным 1.

- Управляемый способ (GUIDED) отличается от динамического тем, что размер назначаемого блока пропорционален количеству еще не распределенных итераций, отнесенному к общему числу нитей в группе, и постоянно уменьшается вплоть до величины, заданной параметром CHUNK_SIZE (последний по порядку блок может включать в себя еще меньше итераций). Как и для динамического способа распределения, по умолчанию параметр CHUNK_SIZE считается равным 1.

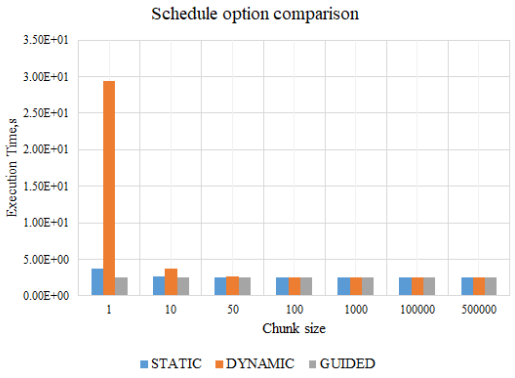
Проведем исследование эффективности распределения нагрузки между нитями в директиве omp for при помощи атрибута schedule для версии параллелизации с использованием reduction для хранения и работы с частичными суммами, зафиксировав число нитей(OMP_NUMB_THREADS=4). Расчеты выполнялись в отсутствие оптимизации.

Chunk size	**Время выполнения,с**			**Ускорение**		
	STATIC	**DYNAMIC**	**GUIDED**	**STATIC**	**DYNAMIC**	**GUIDED**
1	3.7946	29.4457	2.4820	2.5910	0.3339	3.9613
10	2.6900	3.7638	2.4765	3.6550	2.6123	3.9702
50	2.5285	2.6909	2.4766	3.8885	3.6538	3.9700
100	2.5047	2.5777	2.4771	3.9255	3.8143	3.9692
1000	2.4880	2.4885	2.4768	3.9518	3.9509	3.9697
100000	2.4858	2.4828	2.4792	3.9553	3.9601	3.9658
500000	2.4890	2.4839	2.4864	3.9502	3.9583	3.9543

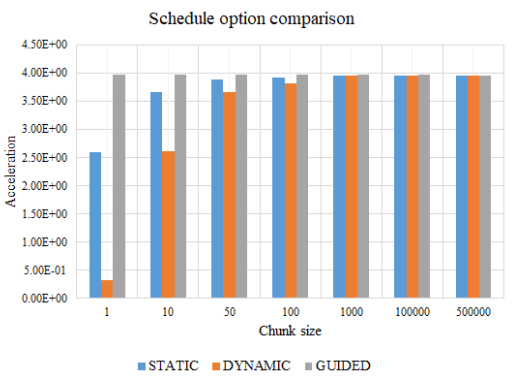
Таблица 9: Исследование применения атрибута schedule в отсутствие оптимизации

Согласно таблице 9 увеличение размера блока итераций вплоть до 1000 влечет к уменьшению времени выполнения программы для способа распределения STATIC. Дальнейшее изменение размера блока практически никак не сказывается на эффективности выполнения программы. Аналогичный результат наблюдается

и при использовании значения атрибута schedule DYNAMIC. Однако, в случае применение способа распределения итераций DYNAMIC существенно больше времени затрачивается на вычисление при малых размерах блока итераций. Это объясняется тем, что к выполнению следующего блока итераций в очереди приступает первая освободившаяся нить, что влияет на равномерность нагруженности нитей. Скорость выполнения программы при значении атрибута schedule GUIDED практически не реагирует на изменение размера блока итераций, что является следствием особенностью перераспределения итераций между блоками. Данные результаты наглядно показаны на рис.11, построенном по данным таблицы 9.



(a) Зависимость времени выполнения от параметра chunk_size



(b) Зависимость ускорения от параметра chunk_size

Рис. 11: Исследование применения атрибута schedule в отсутствие оптимизации

3 Вывод

Проведено исследование возможностей автоматического параллелизма, базовых концепций параллелизации циклов, особенностей работы различных директив синхронизации OpenMP. Получены следующие результаты:

- Использование ключа $-ftree-parallelize-loops=n$ для обеспечения автоматического распараллеливания циклов компиляторов существенно сказывается на эффективности выполнения программы, содержащий операции с массивами, повышая производительность при наличии оптимизации;
- Использование ключа $-ftree-parallelize-loops=n$ не влияет на производительность последовательных программ в случае компиляции программы с ключом -O0(отсутствие оптимизации). Аналогичный эффект наблюдается для программ, где работа с массивами отсутствует для любых ключей оптимизации;
- Параллелизация методом SMPD позволяет более тонко перераспределять нагрузку между нитями при работе с циклами, что в данной работе приводит к наилучшей производительности по отношению к другим способам оптимизации;

- Параллелизация с сохранением результата частичного суммирования в массив показывает наихудший результат, поскольку происходит многократное хаотичное обращение к общей области памяти из нитей, таким образом нарушается целостность кеша;
- Нагрузку между нитями можно настроить в секции omp for автоматического распараллеливания цикла путем задания атрибута schedule;
- Эффективность параллелизации улучшается при увеличение размера блока итераций chunk_size вплоть 1000 для методов STATIC и DYNAMIC, однако дальнейшее увеличение параметра не влечет к увеличению скорости выполнения;
- Метод DYNAMIC показывает наихудшие результаты при малых размерах блока итераций;
- При использование параметра GUIDED время выполнения практически не изменяется с изменением размера блока итераций chunk_size, что связано с особенностью работы данного атрибута;
- В данном случае предпочтительным является использование параметра GUIDED, поскольку при его применении получается максимальное ускорение, относительно выполнения последовательной версии программы при любом размере chunk_size.

4 Приложение

https://github.com/nkmashaev/OpenMP_Lab2 - код программы