Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Параллельные численные методы»

Лабораторная работа №2
Вычисление определенного интеграла

Козинов Е.А., Мееров И.Б., Сысоев А.В.

При поддержке компании Intel

Нижний Новгород
2010

Содержание

Введение

Задача вычисления определенного интеграла I для некоторой заданной на отрезке $[a, b]$ функции $f(x)$ является классической задачей математического анализа. Известно [1], что для функций, имеющих на $[a, b]$ конечное число точек разрыва первого рода, такое значение существует, единственно и может быть формально получено по определению как

$$I = \int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(\xi_i)\Delta x_i ,$$

где $x_0=a$, $x_n=b$, а x_i, $i=1,..,n-1$ – произвольная упорядоченная система точек отрезка $[a, b]$ такая, что $\Delta x_i \to 0$ при $n \to \infty$; ξ_i – произвольная точка отрезка $[x_{i-1}, x_i]$.

В математическом анализе обосновывается аналитический способ нахождения значения интеграла с помощью знаменитой формулы Ньютона-Лейбница [1]

$$\int_a^b f(x)\,dx = F(b) - F(a) ,$$

где $F(x)$ – некоторая первообразная для данной функции $f(x)$. К сожалению, применение этого весьма привлекательного подхода к вычислению I наталкивается на несколько серьезных препятствий.

Во-первых, для многих элементарных функций $f(x)$ не существует первообразной среди элементарных функций: например, отсутствуют первообразные для функций

$$\frac{\sin x}{x}, \quad \frac{1}{\ln x}, \quad e^{-x^2} .$$

Во-вторых, даже если первообразная $F(x)$ для заданной функции $f(x)$ найдена, то вычисление двух ее значений $F(a)$ и $F(b)$ может оказаться более трудоемким, чем вычисление существенно большего количества значений $f(x)$.

И наконец, для многих реальных приложений определенного интеграла характерна дискретность задания подынтегральной функции, что делает указанный аналитический подход не применимым в принципе.

Сказанное предопределяет необходимость использования приближенных формул для вычисления определенного интеграла на основе значений подынтегральной функции $f(x)$. Такие специальные приближенные формулы называют *квадратурными формулами* или *формулами численного интегрирования*. Происхождение термина можно связать с геометрическим смыслом определенного интеграла: вычисление I при $f(x)\geq 0$ равносильно построению квадрата, равновеликого криволинейной трапеции с основанием $[a, b]$ и «крышей» $f(x)$.

Простейшую квадратурную формулу – *формулу прямоугольников* – можно вывести непосредственно из определения интеграла, зафиксировав некоторое n, выбрав равномерную систему точек $x_i = a + ih$ и определив $\xi_i = (x_i + x_{i-1})/2$. Узнать о более сложных квадратурных формулах – семействе формул Ньютона-Котеса, формулах Гаусса и Чебышева – интересующиеся читатели могут, например, в книгах [2, 3].

В данной лабораторной работе мы ограничимся использованием формулы прямоугольников. В работе будут рассмотрены: различные подходы к распараллеливанию метода прямоугольников; идеи по алгоритмической оптимизации, приводящие к уменьшению времени вычислений; методы параллельной отладки.

В силу сочетания сравнительно несложной постановки задачи и метода решения работа носит вводный характер. Вместе с тем в работе сделан акцент на ряд характерных вопросов оптимизации, отладки и распараллеливания программы с использованием пакета программных инструментов Intel Parallel Studio (компилятор, отладчик, профилировщик).

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучение принципов написания высокопроизводительных реализаций алгоритмов с использованием современных компилирующих и отладочных средств.

Данная цель предполагает решение следующих основных задач:

1. Изучение общей схемы алгоритма численного расчета определенного интеграла, обсуждение программной реализации и возможных подходов к ее оптимизации по скорости.
2. Написание нескольких программных реализаций численного расчета определенного интеграла и сравнение их производительности.

3. Демонстрация использования инструментов пакета Intel Parallel Studio в процессе реализации и оптимизации программного кода.
4. Изучение подходов, позволяющих увеличить производительность программных реализаций алгоритмов.
5. Сравнение на данном учебном примере нескольких подходов к распараллеливанию.

1.2. Структура работы

Работа построена следующим образом: дается краткая информация из предметной области – слушателям напоминаются некоторые подходы к численному интегрированию, в частности, дается описание метода прямоугольников. Предлагается тестовая функция, которую невозможно проинтегрировать аналитически. На основе поставленной задачи численного интегрирования тестовой функции демонстрируются подходы к программной реализации метода прямоугольников, а также влияние оптимизирующего компилятора компании Intel на производительность.

На втором этапе предлагается несколько вариантов распараллеливания программной реализации алгоритма. Демонстрируются возможные проблемы, возникающие при написании параллельных программ, а также методы их выявления и преодоления.

Далее в лабораторной работе демонстрируется влияние предварительных вычислений на производительность программной реализации. Показывается, каким образом можно повысить эффективность предварительных вычислений благодаря применению предвычислений и буферизации.

В завершение работы обсуждается влияние алгоритмической оптимизации на производительность численного интегрирования. На данном учебном примере демонстрируется «золотое правило»: алгоритмическая оптимизация чаще всего ведет к большему приросту производительности, чем любая программная оптимизация. Приводится параллельная реализация финальной оптимизированной версии алгоритма.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	2 четырехъядерных процессора Intel Xeon E5520 (2.27 GHz)
Память	16 Gb

Операционная система	Microsoft Windows 7
Среда разработки	Microsoft Visual Studio 2008
Компилятор, профилировщик, отладчик	Intel Parallel Studio SP1

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий.

1. Кратко напомнить студентам о методах численного интегрирования, таких как метод прямоугольников и трапеций. Попросить слушателей вывести основные формулы для вычисления интегралов от функций общего вида в пространствах произвольной размерности.
2. Познакомить слушателей с предлагаемой в лабораторной работе тестовой функцией. Для предлагаемой функции слушатели должны выписать основные вычислительные формулы.
3. Выполнить последовательную программную реализацию алгоритма численного интегрирования методом прямоугольников. Провести вычислительные эксперименты, сравнить полученное решение с «точным»[1] решением. Обратить внимание на корректность реализации.
4. Использовать при сборке приложения оптимизирующий компилятор компании Intel. Сравнить время вычислений и полученные значения интегралов в версиях, собранных Microsoft C++ Compiler и Intel C++ Compiler.
5. Обсудить возможные способы разделения данных для распараллеливания численного интегрирования методом прямоугольников. Реализовать и сравнить рассмотренные подходы к распараллеливанию. Продемонстрировать возможные «параллельные ошибки» и их влияние на результат работы алгоритма. Показать слушателям методы выявления и устранения «параллельных ошибок» с помощью инструментов пакета Intel Parallel Studio.
6. Реализовать алгоритм численного интегрирования тестовой функции с использованием предварительных вычислений. Сравнить полученные значения интегралов и время выполнения вычислений предыдущих ре-

[1] Так как в качестве примера дана функция, не интегрируемая аналитически, за «точное» решение принимается значение, приведенное в лабораторной работе. При сравнении необходимо учитывать ограничения разрядной сетки.

ализаций и новой версии последовательного кода. Показать, как повысить эффективность предварительных вычислений благодаря буферизации.

7. Продемонстрировать повышение эффективности реализации при применении алгоритмической оптимизации.

При наличии группы слушателей со знаниями и навыками, существенно превышающими средние, реализовать в рамках работы несколько алгоритмов численного интегрирования, а также выполнить их сравнительный анализ по производительности и отклонению значения полученного интеграла.

2. Интегрирование по методу прямоугольников

На практике, довольно часто приходится подсчитывать значение определенного интеграла. Например, в задачах физики, если известно распределение плотности тока, общий ток в сети можно узнать, вычислив определенный интеграл. В теории вероятности один из способов задать случайную величину – задать плотность распределения вероятности. Для подсчета вероятности, на практике, также вычисляют определенный интеграл. Методов вычисления интеграла достаточно много. Распространенным способом вычисления определенного интеграла, является применение квадратурных формул, таких как методы прямоугольников или трапеций.

Пусть на отрезке [*a*, *b*] задана непрерывная неотрицательная функция $y = f(x)$. В этом случае значение определенного интеграла от *f*(*x*) на отрезке [*a*, *b*] совпадает с площадью фигуры, ограниченной графиком функции, осью Ox и прямыми $x = a$, $x = b$ (см. рис. 1).

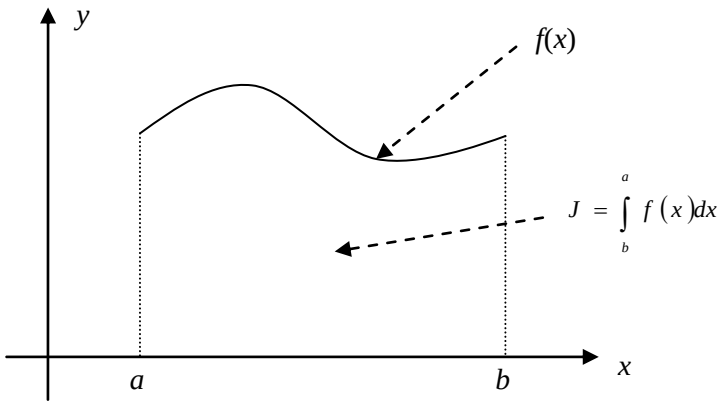

Рис. 1. Геометрический смысл определенного интеграла

При решении прикладных задач приходится сталкиваться с достаточно сложными функциями, которые не интегрируются аналитически. В этом случае значение интеграла вычисляется приближенно. Если функция близка к константе, то значение интеграла можно заменить площадью прямоугольника. Если функция близка к линейной, то интеграл может быть подсчитан через площадь трапеции. Погрешность вычислений квадратурных формул метода прямоугольников[2] может быть вычислена по следующей формуле [3]:

$$\left|R(f)\right| \le \frac{(b-a)^3}{24}\left\|f''\right\|_C$$

Для повышения точности вычисления интеграла отрезок интегрирования разбивается на смежные непересекающиеся отрезки. На каждом отрезке, значение интеграла также может быть заменено на площадь прямоугольника или трапеции. На рис. 2 изображен пример численного вычисления интеграла методами трапеций и прямоугольников.

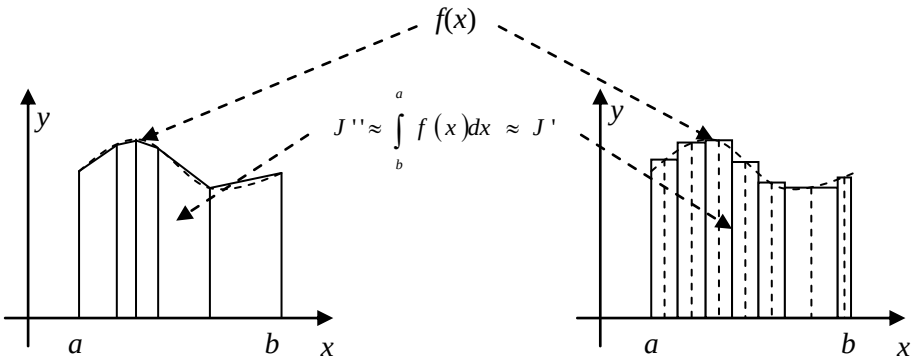

Рис. 2. Численное вычисление интеграла методами трапеций и прямоугольников

Ниже представлен один из вариантов схемы вычисления интеграла методом прямоугольников, когда отрезок интегрирования разбивается на равные части.

[2] Ниже приведенная оценка верна в том случае, когда в качестве значения функции берется середина отрезка $f\left(\frac{a+b}{2}\right)$.

$$\begin{cases} J = \int_b^a f(x)\,dx \approx h\sum_{i=0}^{N-1} f(x_i) \\ x_i = a + ih + \dfrac{h}{2} \end{cases}, \qquad (1)$$

где N – количество отрезков интегрирования, а $h = (b - a) / N$.

Известно, что погрешность вычислений в этом случае может быть оценена следующим неравенством [3]:

$$\left|R(f)\right| \le M_2 \frac{b-a}{24} h^2, \text{ где } M_2 = \max_{x\in[a,b]} \left|f''(x)\right|$$

Если дана двумерная функция $y = f(x, y)$ и область интегрирования D представляет собой прямоугольник $[a_1, b_1] \times [a_2, b_2]$, то приближенное значение интеграла, по аналогии с рассмотренным «одномерным» методом, может быть вычислено методом параллелепипедов по формулам (2).

$$\begin{cases} J = \int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x, y)dxdy \approx h_1 h_2 \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} f(x_i, y_i) \\ x_i = a_1 + ih_1 + \dfrac{h_1}{2},\ y_i = a_2 + ih_2 + \dfrac{h_2}{2} \end{cases}, \qquad (2)$$

где N – количество участков интегрирования по оси x, M – количество участков интегрирования по оси y, $h_1 = (b_1 - a_1) / N$ и $h_2 = (b_2 - a_2) / M$.

Если функция $f(x, y)$ непрерывная и гладкая, то погрешность вычисления интеграла также квадратично зависит от h_1 и h_2:

$$\left|R(f)\right| \le \frac{(b_1 - a_1)(b_2 - a_2)}{24}\left(h_1^2 M_{2x} + h_2^2 M_{2y}\right),$$

где $M_{2x} = \max_{(x,y)\in D}\left|\dfrac{\partial^2 f(x,y)}{\partial x^2}\right|$, $M_{2y} = \max_{(x,y)\in D}\left|\dfrac{\partial^2 f(x,y)}{\partial y^2}\right|$ [4].

В качестве учебного примера рассмотрим тестовую функцию

$$\begin{cases} f(x,y) = \dfrac{e^{\sin(\pi x)\cdot\cos(\pi y)} + 1}{(b_1 - a_1)\cdot(b_2 - a_2)}, \text{где} \\ (x,y) \in [a_1, b_1] \times [a_2, b_2] \\ a_1 = 0, b_1 = 16 \\ a_2 = 0, b_2 = 16 \end{cases}. \quad (3)$$

3. Программная реализация

3.1. Последовательная версия. Базовая реализация алгоритма интегрирования

Программную реализацию начнем с написания функции, выполняющей расчет приближенного значения интеграла для учебного примера (3) по формулам (2). Здесь же продемонстрируем основные действия по созданию проекта в среде **Microsoft Visual Studio 2008**.

Прежде всего, создадим новое **Решение** (**Solution**), в которое включим первый **Проект** (**Project**) данной лабораторной работы. Последовательно выполните следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2008**.
- В меню **File** выполните команду **New→Project…**.
- Как показано на рис. 3, в диалоговом окне **New Project** в типах проекта выберите **Win32**, в шаблонах **Win32 Console Application**, в поле **Solution** введите **02_Integral**, в поле **Name** – **01_Reference**, в поле **Location** укажите путь к папке с лабораторными работами курса – **c:\ParallelCalculus**. Нажмите **OK**.

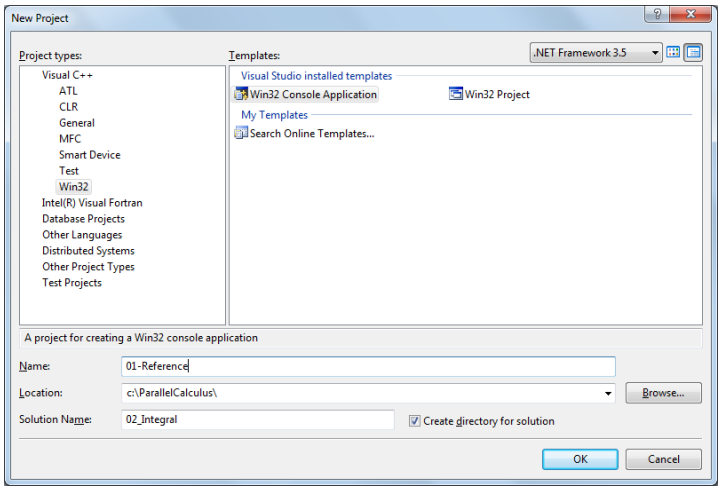

Рис. 3. Создание решения для лабораторной работы

- В диалоговом окне **Win32 Application Wizard** нажмите **Next** (или выберите **Application Settings** в дереве слева) и установите флаг **Empty Project**. Нажмите **Finish**.
- В окне **Solution Explorer** в папке **Source Files** выполните команду контекстного меню **Add→New Item…**. В дереве категорий слева выберите **Code**, в шаблонах справа – **C++ File (.cpp)**, в поле **Name** введите имя файла **main_s**. Нажмите **Add**.

В результате выполненной последовательности действий в окне редактора кода **Visual Studio** будет открыт пустой файл **main_s.cpp**.

Далее создадим заготовку функции **`main()`**. Функция должна содержать код, несколько раз запускающий тестируемую реализацию алгоритма и вычисляющий минимальное, максимальное и среднее времена ее работы.

```
int main ()
{
  // переменная цикла
  int i;

  // время проведенного эксперимента
  double time;
  // значение вычисленного интеграла
  double res;
  // минимальное время работы реализации алгоритма
  double min_time;
  //максимальное время работы реализации алгоритма
  double max_time;
  // среднее время работы реализации алгоритма
  double avg_time;
```

```
  // количество запусков программы
  int numbExp = 10;

  // первый запуск
  min_time = max_time = avg_time = experiment(&res);
  // оставшиеся запуски
  for(i = 0; i < numbExp - 1; i ++)
  {
    time = experiment(&res);
    avg_time += time;
    if(max_time < time) max_time = time;
    if(min_time > time) min_time = time;
  }
  // вывод результатов эксперимента
  printf("integral value : %lf; \n", res);
  printf("execution time : %lf; %lf; %lf \n",
    avg_time / numbExp, min_time, max_time);

  return 0;
 }
```

Далее необходимо реализовать функцию **experiment()**. Данная функция через параметр должна возвращать значение подсчитанного интеграла. Результатом выполнения функции должно быть время работы программной реализации алгоритма интегрирования. Внутри функции должны быть заданы параметры экспериментов. Также функция должна содержать код, вызывающий реализацию алгоритма интегрирования и замеряющий время ее работы. Отметим, что здесь и во всех остальных функциях мы используем вещественные числа с двойной точностью, то есть тип **double**. Код, который необходимо написать, представлен ниже.

```
double experiment(double *res)
{
  double stime, ftime; // время начала и конца рассчета
  double a1 = 0.0 ;    // левая граница интегрирования
                       //  по координате x
  double b1 = 16.0;    // правая граница интегрирования
                       //  по координате x
  double a2 = 0.0 ;    // левая граница интегрирования
                       //  по координате y
  double b2 = 16.0;    // правая граница интегрирования
                       //  по координате y
  double h = 0.001;    // шаг интегрирования

  stime = omp_get_wtime( );
  // вызов функции интегрирования
  integral(a1, b1, a2, b2, h, res);
  ftime = omp_get_wtime( );
```

```
  return (ftime - stime);
}
```

Следующим шагом напишем функцию **`integral()`** для вычисления приближенного значения интеграла тестовой функции (3) по формулам (2). Первые четыре параметра функции – пределы интегрирования. Пятый параметр – шаг сетки интегрирования (предполагается, что $h_1 = h_2 = h$). Последним параметром функция должна возвращать значение подсчитанного интеграла.

Возможный вариант реализации алгоритма численного интегрирования по формуле (2) представлен в следующем листинге.

```
void integral(const double a1, const double b1,
  const double a2, const double b2, const double h,
  double *res)
{
    int i, j, n1, n2;
    double sum;       // локальная переменная
                      //   для подсчета интеграла
    double x;         // координата точки сетки по оси x
    double y;         // координата точки сетки по оси y

    // количество точек сетки интегрирования
    //  n1 - по координате x
    //  n2 - по координате y
    n1 = (int)((b1 - a1) / h);
    n2 = (int)((b2 - a2) / h);

    sum = 0.0;
    for(i = 0; i < n1; i++)
    {
        for(j = 0; j < n2; j++)
        {
            // вычисление координат точек
            x = a1 + i * h + h / 2;
            y = a2 + j * h + h / 2;
            // вычисление интеграла
            sum += ((exp(sin(x * PI) * cos(y * PI)) + 1) /
                   ((b1 - a1) * (b2 - a2))) * h * h;
        }
    }
    *res = sum;
}
```

Самое последнее действие, которое осталось выполнить до перехода к следующей части лабораторной работы, – собрать получившийся код (команда **Build→Build Solution**) и запустить его на выполнение.

Убедитесь, что вывод программы соответствует рис. 4.

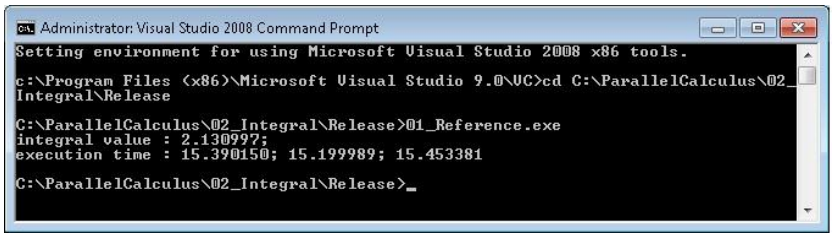

Рис. 4. Результаты базовой версии кода

3.2. Последовательная версия. Эффект применения компилятора Intel Compiler

Для того чтобы уменьшить время работы программной реализации алгоритма, существует большое количество методик. Самое простое, что можно сделать, – использовать компилятор, генерирующий наиболее оптимальный код под вашу аппаратную платформу.

Как известно, разные компиляторы генерируют исполняемый код различного качества. Наиболее простая ситуация, имеющая место, например, при сборке в конфигурации **Debug**, состоит в том, что код с языка высокого уровня переводится в машинный так, что он выполняет ровно те действия и ровно в том порядке, как они указаны в исходном коде. Максимальной производительности в этом случае добиться невозможно хотя бы потому, что не учитываются многие архитектурные особенности процессора[3]. Оптимизирующие компиляторы (при сборке в конфигурации **Release**, то есть с ключами оптимизации) существенно перестраивают код в процессе трансляции с целью задействовать максимальное количество доступных возможностей процессора, позволяющих ускорить вычисления. В лабораторной работе используется один из лучших оптимизирующих компиляторов –**Intel C/C++ Compiler**.

На данном этапе лабораторной работы предлагается попробовать использовать оптимизирующий компилятор **Intel C/C++ Compiler**, являющийся частью **Parallel Studio**. Для этого в окне **Solution Explorer** выберите проект **01_Reference** и выполните команду контекстного меню **Intel Parallel Composer→Use Intel C++…**. В диалоговом окне **Confirmation** нажмите **OK**.

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение. Результаты вычисления интеграла не должны измениться, но при этом время вычислений должно резко сократиться.

---

[3] Заметим, что ничего удивительного в этом нет. Конфигурация Debug предназначена для отладки программы. Пока программа работает неправильно, речи о ее оптимизации идти не может.

Убедитесь, что вывод программы соответствует рис. 5.

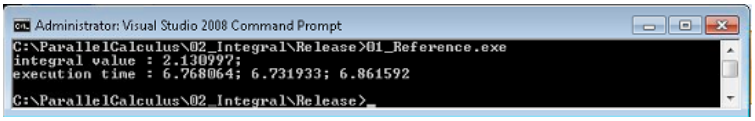

Рис. 5. Результаты базовой версии кода (компилятор **Intel C/C++ Compiler**)

На графике, представленном на рис. 6, показано сравнение времени выполнения алгоритма численного интегрирования с использованием разных компиляторов.

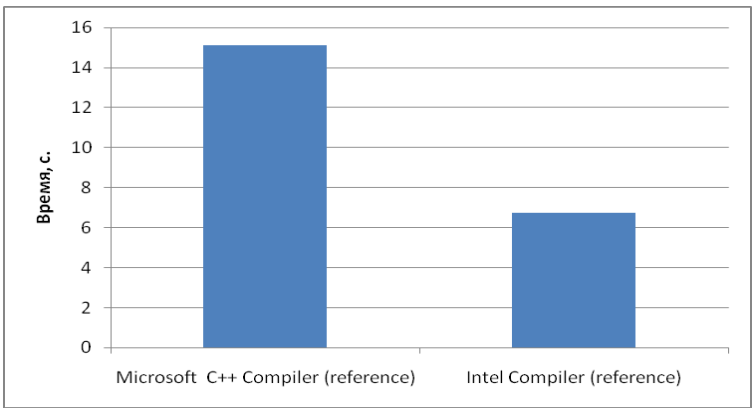

Рис. 6. Сравнение времени базовой версии кода (компиляторы Microsoft **C++ Compiler** и **Intel C/C++ Compiler**)

В последующих экспериментах будет применяться только компилятор **Intel C/C++ Compiler**, таким образом, за точку отсчета возьмем времена, представленные на рис. 5.

3.3. Параллельная версия. Распараллеливание базового алгоритма

Посмотрим на загрузку процессора во время выполнения базовой реализации (рис. 7). Учитывая параметры тестовой инфраструктуры, представленные в § 1.3, можно сказать, что в процессе вычислений участвует только одно ядро из имеющихся восьми.

Следующим шагом предлагается распараллелить базовую реализацию алгоритма. Один из наиболее простых подходов – геометрическая декомпозиция данных. Данный подход предполагает разделение данных на части и применение к ним одного и того же алгоритма. В численном интегрировании мы имеем дело с прямоугольной сеткой, в каждом узле которой вычисляется функция и умножается на квадрат шага. Вычисление функции в одном узле сетки не зависит от соседних узлов, таким образом, поделив сетку между потоками, можно получить параллельную версию, причем

ожидаемое ускорение должно быть близко к линейному. Далее мы рассмотрим несколько параллельных версий алгоритма с разными способами разделения данных.

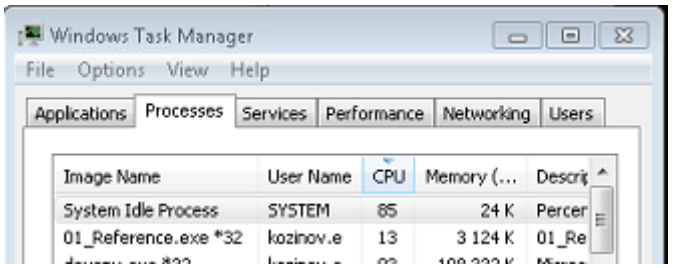

Рис. 7. Загрузка процессора при исполнении базовой реализации

3.3.1. Демонстрация гонки данных и методов их обнаружения

В системах с общей памятью основной способ распараллеливания – использование многопоточности. Существуют различные реализации механизмов работы с потоками. Наиболее известные – **p_thread** под операционными система семейства **Linux** и **Windows Threads** под ОС **Windows** соответственно. К сожалению, применение потоков в явном виде часто является непростой задачей. Для упрощения написания параллельных программ, использующих потоки, часто применяют технологию OpenMP.

Используя средства OpenMP, реализуем первую параллельную версию подсчета интеграла с разделением сетки интегрирования по столбцам (рис. 8).

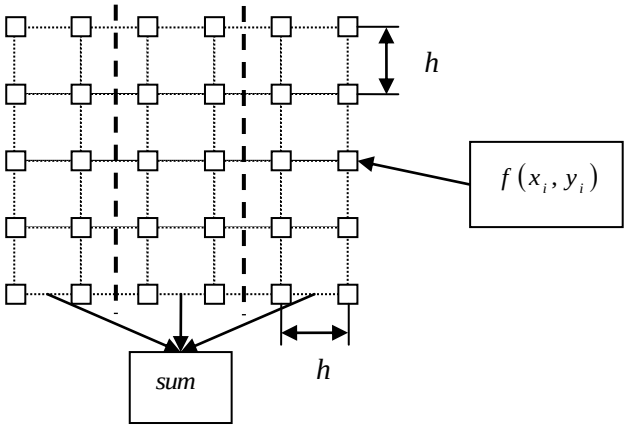

Рис. 8. Схема алгоритма с разделением данных по столбцам

Прежде всего, создадим в рамках решения **02_Integral** новый проект с названием **02_Integral_col**. Повторите все действия, описанные в § 3.1, с

той лишь разницей, что начать нужно с выбора решения **02_Integral** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_col**.

После получения пустого файла **main_col.cpp** скопируем в него код из файла **main_s.cpp** проекта **01_Reference**.

Затем выполним переход к использованию компилятора **Intel C++**.

Наконец, настроим в свойствах проекта использование OpenMP. В дереве **Configuration Properties** перейдите к разделу **C/C++→Language** и в поле **OpenMP Support** справа выберите вариант: **Generate Parallel Code (/openmp, equiv. to /Qopenmp)**.

Согласно схеме разбиения данных, представленной на рис. 8, необходимо распараллеливать внешний цикл функции **`integral()`**. Применим директиву OpenMP **`omp parallel for`**, позволяющую создать потоки и распределить итерации цикла между ними.

```
//разделение точек сетки интегрирования по оси x
#pragma omp parallel for
for(i = 0; i < n1; i++)
{
    for(j = 0; j < n2; j++)
    {
        //вычисление координат точки
        x = a1 + i * h + h / 2;
        y = a2 + j * h + h / 2;
        //вычисление интеграла
        sum += ((exp(sin(x * PI) * cos(y * PI)) + 1) /
               ((b1 - a1) * (b2 - a2))) * h * h;
    }
}
```

Соберите проект **02_Integral_col** и запустите на исполнение. Убедитесь, что на многоядерной/многопроцессорной системе результат работы функции подсчета интеграла существенно отличается от последовательного и, кроме того, меняется от запуска к запуску (см. рис. 9).

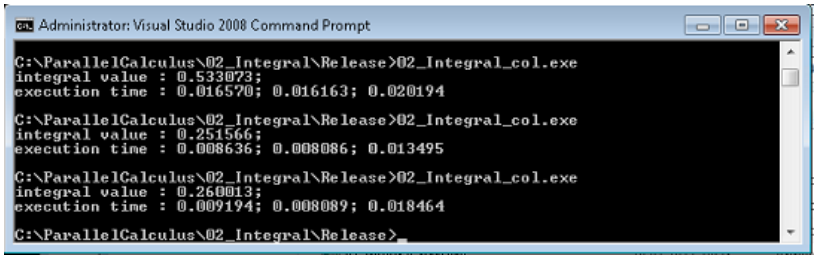

Рис. 9. Гонки данных

В чем же дело? Ответить на этот вопрос может помочь инструмент **Intel Parallel Inspector** из пакета **Intel Parallel Studio**. Данный инструмент позволяет выявить ошибки, как в работе с памятью, так и с потоками. **Intel Parallel Inspector** является дополнением к среде разработки **Microsoft Visual Studio**. После установки инструмента в среде появляется панель инструментов, изображенная на рис. 10.

Используя выпадающий список, разработчик может выбрать, какой тип ошибок он хочет диагностировать. В данном случае нас интересует причина некорректных результатов параллельной реализации. Следовательно, необходимо выбрать анализ ошибок многопоточности (вариант **Threading errors**) и нажать на значок запуска **Inspect**. После нажатия появится диалоговое окно настройки, в котором необходимо выбрать уровень качества поиска ошибок, после чего произвести запуск анализа кода (см. рис. 11).

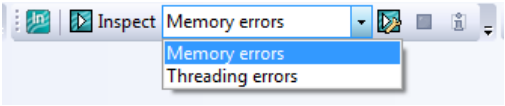

Рис. 10. Панель инструментов Intel Parallel Inspector

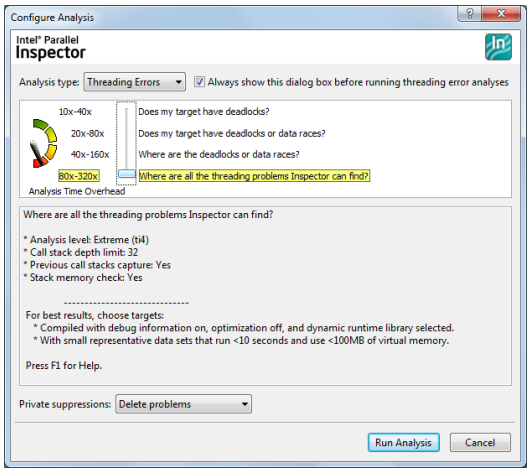

Рис. 11. Окно конфигурации Intel Parallel Inspector

Выберите максимальный уровень анализа кода и произведите запуск программы. После завершения работы будет выведен отчет, из которого следует, что добавление директивы `omp parallel for` привело к большому количеству ошибок, являющихся гонками данных. Если анализируемый код был собран в режиме **Debug**, можно увидеть, где произошла ошибка вплоть до строчки (см. рис. 12).

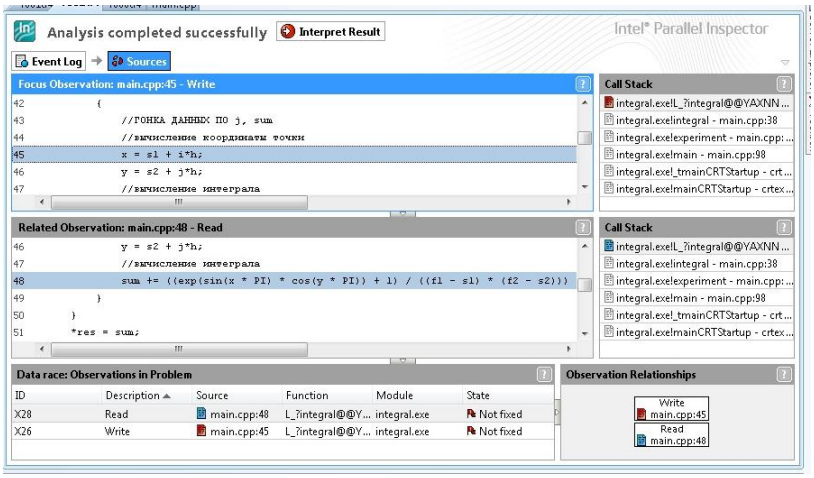

Рис. 12. Результат анализа кода

Просмотрите отчет, полученный **Intel Parallel Inspector**, и проанализируйте представленные в нем ошибки.

3.3.2. Разделение данных по столбцам

Метод разделения данных по столбцам был описан в предыдущем разделе. Параллельная версия, полученная простым добавлением директивы **`omp parallel for`**, содержит, как мы выяснили, гонки данных. Ошибки связаны с четырьмя переменными:

- **`x`** и **`y`** – координаты точки, где необходимо вычислить функцию;
- **`j`** – переменная внутреннего цикла;
- **`sum`** – переменная для накопления суммы интеграла.

Для того чтобы исправить ошибки, первые три переменные необходимо сделать локальными для потоков, а по переменной **`sum`** организовать редукцию данных с операцией суммирования. Для локализации данных в директиве **`parallel`** можно использовать параметр **`private`**(**`<список переменных>`**). Для редукции данных в директиве **`omp for`** используется параметр **`reduction`**(**`<операция>`**:**`<список переменных>`**).

Произведите изменения кода, связанные с исправлением ошибок многопоточности. В результате код численного нахождения интеграла должен приобрести вид, представленный в следующем листинге.

```
// Сетка интегрирования делится по столбцам
// разделение точек сетки интегрирования по оси x
#pragma omp parallel for private (x, y, j) /
  reduction(+: sum)
for(i = 0; i < n1; i++)
{
```

```
  for(j = 0; j < n2; j++)
  {
    //вычисление координат точки
    x = a1 + i * h + h / 2;
    y = a2 + j * h + h / 2;
    //вычисление интеграла
    sum += ((exp(sin(x * PI) * cos(y * PI)) + 1) /
           ((b1 - a1) * (b2 - a2))) * h * h;
  }
}
```

Проанализируйте полученный код с помощью **Intel Parallel Inspector**. Результат анализа должен показать, что в полученной версии кода гонок данных нет. Пример полученного отчета представлен на рис. 13.

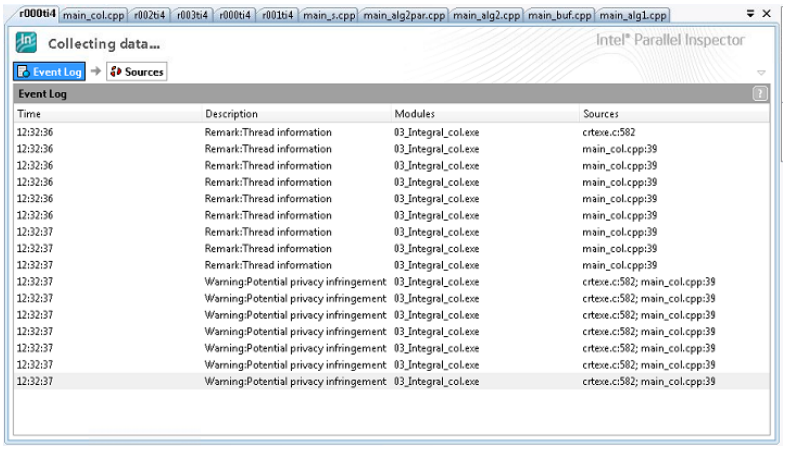

Рис. 13. Отчет **Intel Parallel Inspector** при отсутствии «параллельных» ошибок

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение. Сравните полученное значение вычисленного интеграла со значением, полученным в последовательной версии. При сравнении следует учитывать, что точное совпадение результатов последовательной и параллельной программы бывает довольно редко – фактически, его можно обеспечить лишь в случае, когда совпадают не только все выполняемые в этих вариантах операции, но и порядок их исполнения[4]. Добиться этого очень непросто, поскольку в многопоточной программе порядок исполнения потоков не известен (что, собственно, и ведет к самой типовой «параллельной» ошибке – гонке данных). Еще одним усложняю-

[4] Исключением являются программы, где не выполняются вещественные вычисления, во-первых, а вычисления с целыми числами не выходят за разрядную сетку используемых типов данных, во-вторых. В этом случае от порядка операций результат не зависит.

щим фактором является, как ни странно, компилятор. Дело в том, что сборка в конфигурации **Release** предполагает оптимизацию кода, часто связанную с существенными изменениями, вносимыми в процессе компиляции в объектный код. В силу большей сложности параллельного кода, компилятор может в нем оптимизировать далеко не все и не так, как в коде последовательном. Тем не менее, в нашем случае, благодаря квадратичной сходимости алгоритма относительно шага, при уменьшении шага значения последовательной и параллельной версии должны стремиться друг к другу. При уменьшении шага также следует учитывать ограничения, накладываемые на вещественные числа. Слишком маленький шаг в свою очередь может вносить погрешности в вычисления.

Убедитесь, что вывод программы соответствует представленному на рис. 14.

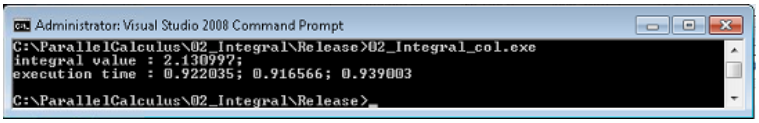

Рис. 14. Результат параллельного вычисления интеграла с разделением данных по столбцам

На графике, представленном на рис. 15, показано сравнение времени выполнения алгоритма численного интегрирования в последовательной и параллельной реализациях.

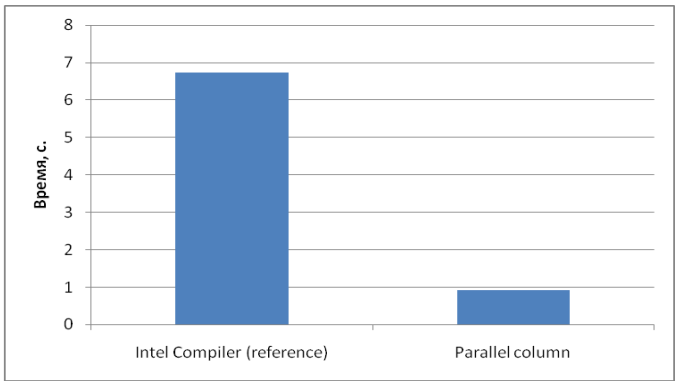

Рис. 15. Сравнение времени численного интегрирования для последовательной и параллельной реализации

3.3.3. Разделение данных по строкам

Сетку интегрирования можно разделять не только по столбцам, но и по строкам. Схематичное изображение метода распараллеливания по строкам представлено на рис. 16.

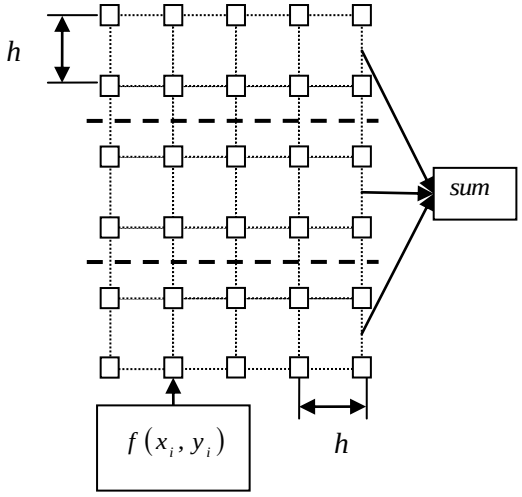

Рис. 16. Схема алгоритма с разделением данных по строкам

Для организации разделения данных по строкам необходимо распараллелить внутренний цикл подсчета интеграла. При реализации сразу учтем ошибки, которые были допущены при написании параллельной версии с разделением данных по столбцам.

Создадим в рамках решения **02_Integral** новый проект с названием **02_Integral_row**. Затем выполним переход к использованию компилятора **Intel C++** и в свойствах проекта включим поддержку использования OpenMP.

В рамках созданного проекта реализуйте параллельный алгоритм численного интегрирования с разделением сетки по строкам.

```
//Сетка интегрирования делится по строкам
for(i = 0; i < n1; i++)
{

  //разделение точек сетки интегрирования по оси y
  #pragma omp parallel for private (x, y) reduction(+: sum)
  for(j = 0; j < n2; j++)
  {
    //вычисление координат точки
    y = a2 + j * h + h / 2;
    x = a1 + i * h + h / 2;
    //вычисление интеграла
    sum += ((exp(sin(x * PI) * cos(y * PI)) + 1) /
           ((b1 - a1) * (b2 - a2))) * h * h;
  }
}
```

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение. Результат вычисления интеграла должен быть близок к последовательной версии.

Убедитесь, что вывод программы соответствует представленному на рис. 17.

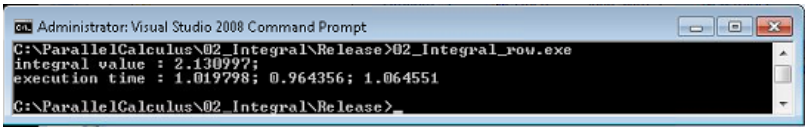

Рис. 17. Результат параллельного вычисления интеграла с разделением данных по строкам

Можно заметить, что время работы стало чуть большим, чем при распараллеливании с разделением сетки интегрирования по столбцам. Объясняется это тем, что в данной реализации существенно больше накладных расходов. Каждый раз, когда открывается параллельная секция OpenMP, дополнительные потоки выводятся из состояния сна. При закрытии параллельной секции дополнительные потоки «засыпают». На графике, представленном на рис. 18, показано сравнение времени выполнения алгоритма численного интегрирования с использованием разных подходов к распределению данных между потоками.

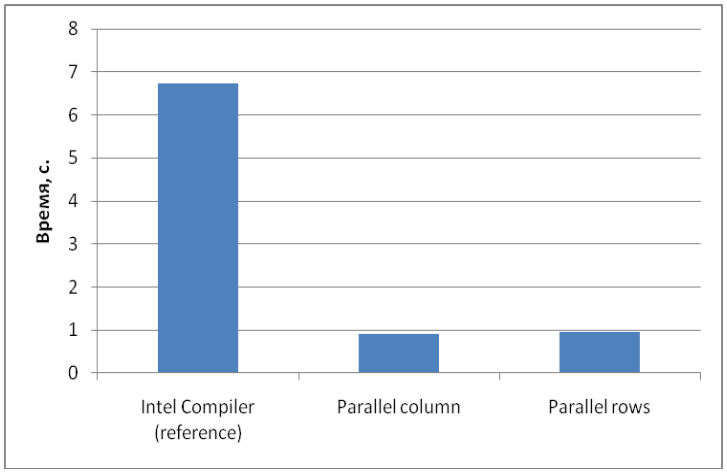

Рис. 18. Сравнение времени численного интегрирования для последовательной и параллельной реализаций

3.3.4. Блочное разделение данных

Еще один популярный способ геометрического разделения данных – разделение на блоки. Данный подход часто хорошо показывает себя при обработке больших массивов данных за счет более эффективного использования кеш-памяти.

Схематичное изображение параллельного алгоритма численного интегрирования с блочным разделением данных представлено на рис. 19.

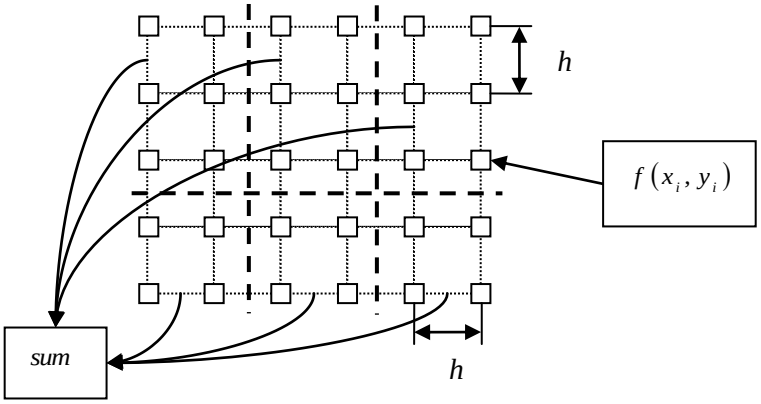

Рис. 19. Схема алгоритма с блочным разделением данных

Создадим в рамках решения **02_Integral** новый проект с названием **02_Integral_block**. Затем выполним переход к использованию компилятора **Intel C++** и в свойствах проекта включим поддержку использования OpenMP.

Блочная схема легко реализуется с помощью использования вложенного параллелизма в OpenMP. Чтобы использовать вложенный параллелизм, необходимо сначала включить его поддержку с помощью вызова библиотечной функции:

```
// включаем возможность использования вложенного
// параллелизма
omp_set_nested(true);
```

Вызов представленной функции должен быть сделан *вне параллельной секции* до использования вложенного параллелизма.

Реализуйте параллельный алгоритм численного интегрирования с разделением сетки интегрирования на блоки. Для этого необходимо распараллелить оба цикла вычисления интеграла.

```
omp_set_nested(true);

// Данные делятся на блоки и для каждого
// блока подсчитывается часть интеграла
// разделение точек сетки интегрирования по оси x
#pragma omp parallel for
For (i = 0; i < n1; i++)
{
  //разделение точек сетки интегрирования по оси y
```

```
  #pragma omp parallel for private (x, y) reduction(+: sum)
  for(j = 0; j < n2; j++)
  {
    //вычисление координат точки
    x = a1 + i * h + h / 2;
    y = a2 + j * h + h / 2;
    //вычисление интеграла
    sum += ((exp(sin(x * PI) * cos(y * PI)) + 1) /
           ((b1 - a1) * (b2 - a2))) * h * h;
  }
}
```

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение. Результат вычисления интеграла должен быть близок к последовательной версии.

Убедитесь, что вывод программы соответствует представленному на рис. 20.

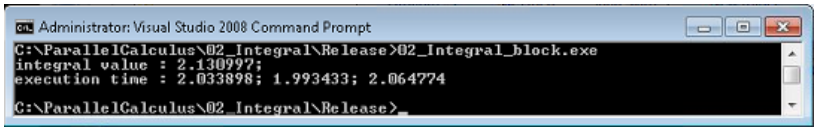

Рис. 20. Результат параллельного вычисления интеграла с блочным разделением данных

Можно заметить, что время работы снова выросло. Причина в еще больших накладных расходах. Однако, как уже было сказано выше, подход имеет право на существование и часто приводит к хорошим результатам. При большем размере задачи, а, значит, при увеличении сложности параллельного участка кода накладные расходы могут быть компенсированы вычислениями.

На графике, представленном на рис. 21, показано сравнение времени выполнения алгоритма численного интегрирования с использованием рассмотренных подходов к распределению данных между потоками.

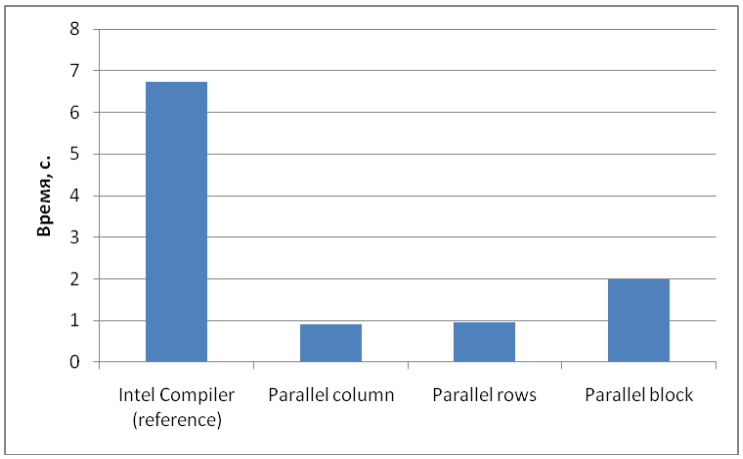

Рис. 21. Сравнение времени численного интегрирования для последовательной и параллельных реализаций

3.4. Последовательная версия. Использование предварительных вычислений сложных функций

Несмотря на достаточно неплохое ускорение, полученное в версии с разделением данных по столбцам, вернемся к вопросу, как еще можно увеличить производительность программы? В общем случае, чтобы ответить на данный вопрос, необходимо найти участки кода, которые нуждаются в оптимизации и занимают основное время вычислений. Для поиска таких мест в пакете **Intel Parallel Studio** имеется инструмент **Intel Parallel Amplifier**. Данный инструмент, как и **Inspector**, является дополнением к среде **MS Visual Studio**. Соответствующая панель инструментов представлена на рис. 22.

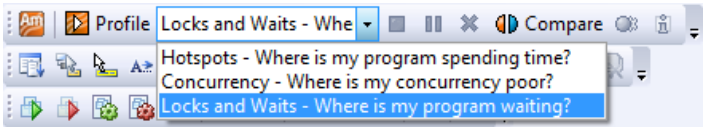

Рис. 22. Панель инструментов Intel Parallel Amplifier

Для профилировки необходимо выбрать первый пункт – поиск «горячих точек». Будем профилировать исходную последовательную версию алгоритма. Запустите **Intel Parallel Amplifier**. Результатом запуска должно стать появление списка горячих точек (рис. 23).

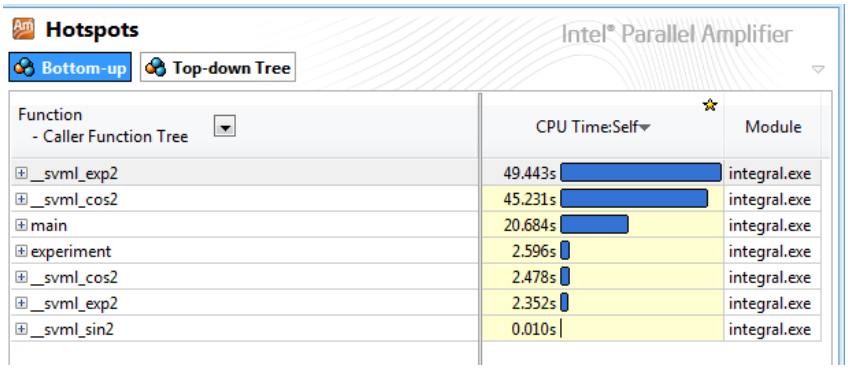

Рис. 23. Список «горячих точек» последовательной реализации

Из результатов анализа видно, что основное время в работе программы занимает вычисление математических функций[5].

Посмотрим внимательно на код численного интегрирования. Можно заметить следующий факт. Значения математических функций **`sin`** и **`cos`** вычисляются много раз. При этом точки, в которых вычисляются эти функции, очень часто повторяются. Для каждой строки сетки интегрирования используется одно и то же значение **`cos`**, а для каждого столбца сетки интегрирования – одно и то же значение **`sin`**. Сами функции **`sin`** и **`cos`** достаточно сложны вычислительно. Исходя из сказанного, повысить эффективность реализации можно, предварительно подсчитав все необходимые значения **`sin`** и **`cos`**, а затем используя их.

Создадим в рамках решения **02_Integral** новый проект с названием **03_AlgOptV1**. Повторите все действия, описанные в § 3.1, с той лишь разницей, что начать нужно с выбора решения **02_Integral** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_alg1**.

Начнем с последовательной версии.

После получения пустого файла **main_alg1.cpp** скопируем в него код из файла **main_s.cpp** проекта **01_Reference**. Затем выполним переход к использованию компилятора **Intel C++**.

Модифицируйте код согласно рассуждениям, описанным в начале параграфа. В начале функции **`integral()`**, необходимо объявить все требуемые переменные.

```
void integral(const double a1, const double b1,
```

[5] Заметим, что во многом поэтому Intel C++ Compiler существенно «переиграл» Microsoft C++ Compiler в данной задаче. Высокопроизводительные реализации математических функций – визитная карточка компиляторов Intel.

```
  const double a2, const double b2, const double h,
  double *res)
{
  int i, j, n1, n2;
  double sum;// локальная переменная для подсчета интеграла
  double x;  // координата точки сетки по оси x
  double y;  // координата точки сетки по оси y
  double *sinx; // значение sin(x * pi)
  double *cosy; // значение cos(y * pi)

  //количество точек сетки интегрирования
  // n1 - по координате x
  // n2 - по координате y
  n1 = (int)((b1 - a1) / h);
  n2 = (int)((b2 - a2) / h);
```

Далее предварительно вычислим значения функции **sin**, используемые в последующих расчетах.

```
  // вычисление значений sin(x * pi)
  sinx = new double [n1];
  for(i = 0; i < n1; i++)
  {
    x = a1 + i * h + h / 2;
    sinx[i] = sin(x * PI);
  }
```

Аналогичным образом вычислим значения функции **cos**.

```
  // вычисление значений cos(y * pi)
  cosy = new double [n2];
  for(j = 0; j < n2; j++)
  {
    y = a2 + j * h + h / 2;
    cosy[j] = cos(y * PI);
  }
```

Последним шагом вычислим значение интеграла с использованием результатов предвычислений.

```
  // вычисление интеграла
  sum = 0.0;
  for(i = 0; i < n1; i++)
  {
    for(j = 0; j < n2; j++)
    {
      // вычисление интеграла
      // (значение sin и cos уже подсчитанны)
      sum += ((exp(sinx[i] * cosy[j]) + 1) /
             ((b1 - a1) * (b2 - a2))) * h * h;
    }
```

```
  }
  *res = sum;
  delete [] sinx;
  delete [] cosy;
}
```

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение. Результаты вычисления интеграла должны остаться прежними (однако см. соображения выше по поводу порядка вычислений, ограничений разрядной сетки и вызванных ими возможных проблем), но время вычислений должно существенно сократиться.

Убедитесь, что вывод программы соответствует представленному на рис. 24.

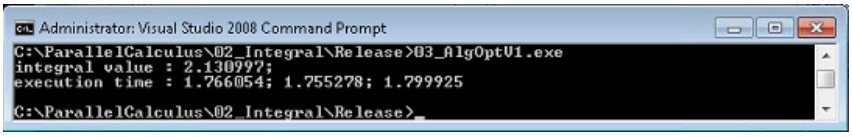

Рис. 24. Результаты последовательной версии с использованием предварительных вычислений

Проанализируйте полученный код на предмет нахождения горячих точек. Результат анализа представлен на рис. 25.

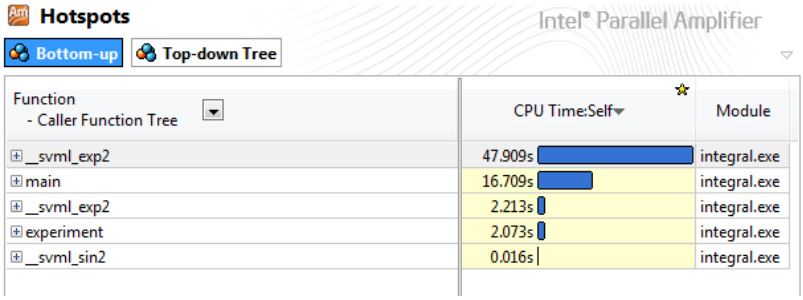

Рис. 25. Список «горячих точек» последовательной реализации, использующей предварительные вычисления

Из полученного профиля видно, что в новой версии выделяется только одна математическая функция, которая работает большую часть времени вычисления интеграла – **`exp`**. В текущей версии кода мы устранили найденные выше горячие точки – функции **`sin`** и **`cos`**, тем самым существенно сократив время вычислений.

На графике, представленном на рис. 26, показано сравнение времени выполнения алгоритма с предварительными вычислениями и других реализаций численного интегрирования.

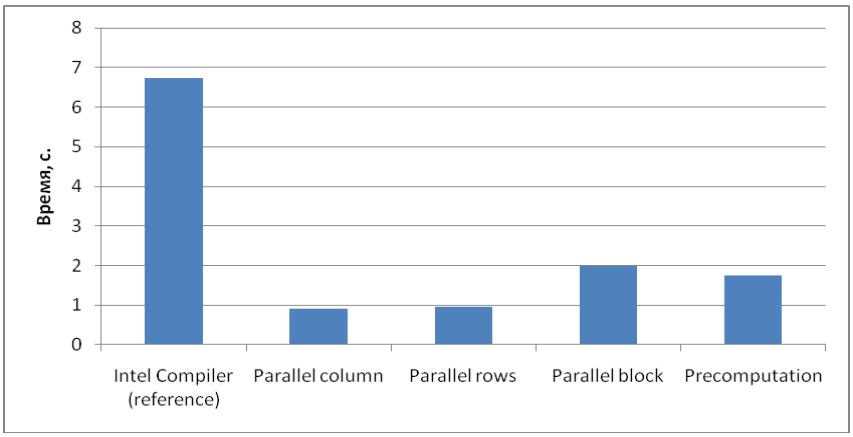

Рис. 26. Сравнение времени численного интегрирования последовательных и параллельных реализаций

Отметим, что время работы базовой последовательной версии более чем в три раза превышает время версии с предварительными вычислениями. Полученные результаты демонстрируют эффективность алгоритмической оптимизации.

3.5. Последовательная версия. Использование предварительных вычислений и буферизации для ускорения вычислений

Недостаток предыдущей реализации заключается в том, что при увеличении области интегрирования резко возрастают расходы на память для хранения подсчитанных значений. В некоторый момент может наступить ситуация, при которой выгода от предварительных вычислений функций перестанет превышать накладные расходы на доступ к памяти. Для решения этой проблемы можно воспользоваться буферизацией.

В данной задаче организовать буферизацию достаточно просто. Для этого необходимо разбить область вычислений на прямоугольники равных размеров. Для каждого прямоугольника подсчитать интеграл и полученные значения суммировать. Размер буфера обычно подбирают таким образом, чтобы количество точек сетки интегрирования делилось на него без остатка.

Реализуем алгоритм вычисления интеграла для тестовой функции с использованием предварительных вычислений и буферизацией. Создадим в рамках решения **02_Integral** новый проект с названием **04_Buf**. Повторите все действия, описанные в § 3.1, с той лишь разницей, что начать нужно с выбора решения **02_Integral** в окне **Solution Explorer** и выполнения ко-

манды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_buf**.

После получения пустого файла **main_Buf.cpp** скопируем в него код из файла **main_alg1.cpp** проекта **01_Reference**. Затем выполним переход к использованию компилятора **Intel C++**.

Объявим необходимые переменные.

```
// размер буфера
#define BUF_SIZE 2000

void integral(const double a1, const double b1,
  const double a2, const double b2, const double h,
  double *res)
{
  int i, j, ii, jj, n1, n2, nb1, nb2;
  double sum;// локальная переменная для подсчета интеграла
  double x;  // координата точки сетки по оси x
  double y;  // координата точки сетки по оси y
  double *sinx; // значение sin(x * pi)
  double *cosy; // значение cos(y * pi)
```

Подсчитаем необходимые размеры сеток и проверим тот факт, что выбран буфер правильного размера[6].

```
  // количество точек сетки интегрирования
  // n1 - по координате x
  // n2 - по координате y
  n1 = (int)((b1 - a1) / h);
  n2 = (int)((b2 - a2) / h);

  // правильность размера блока
  assert((n1 % BLOCK_SIZE) == 0);
  assert((n2 % BLOCK_SIZE) == 0);

  // Вычисление количества точек в блоке
  // nb1 - по координате x
  // nb2 - по координате y
  nb1 = n1 / BLOCK_SIZE;
  nb2 = n2 / BLOCK_SIZE;
```

Выделим необходимую память.

```
  // Выделение памяти
  sinx = new double [BLOCK_SIZE];
  cosy = new double [BLOCK_SIZE];
```

[6] Желающие могут модифицировать программу для случая, когда размер сетки не кратен размеру буфера.

Далее требуется пройти в цикле по всем блокам и вычислить значение интеграла. Полученные значения сложить. Как и в предыдущем разделе для ускорения вычисления значения интеграла будем использовать предварительные вычисления, но уже для блока.

```
  // проход по всем блокам
  for(ii = 0; ii < n1; ii += BLOCK_SIZE)
  {
    // вычисление значений sin(x * pi)
    for(i = 0; i < BLOCK_SIZE; i++)
    {
      x = a1 + i * h + ii + h / 2;
      sinx[i] = sin(x * PI);
    } /* for(i = 0; i < BLOCK_SIZE; i++) */
    for(jj = 0; jj < n2; jj += BLOCK_SIZE)
    {
      // вычисление значений cos(y * pi)
      for(j = 0; j < BLOCK_SIZE; j++)
      {
        y = a2 + j * h + jj + h / 2;
        cosy[j] = cos(y * PI);
      } /* for(j = 0; j < BLOCK_SIZE; j++) */
      // вычисление интеграла
      for(i = 0; i < BLOCK_SIZE; i++)
      {
        for(j = 0; j < BLOCK_SIZE; j++)
        {
          // вычисление интеграла
          sum += ((exp(sinx[i] * cosy[j]) + 1) /
                 ((b1 - a1) * (b2 - a2))) * h * h;
        } // for(j = 0; j < BLOCK_SIZE; j++)
      } // for(i = 0; i < BLOCK_SIZE; i++)
    } // for(jj = 0; jj < n2; jj += BLOCK_SIZE)
  } // for(ii = 0; ii < n1; ii += BLOCK_SIZE)
```

В финале нужно вернуть значение подсчитанного интеграла и освободить выделенную память.

```
  *res = sum;
  delete [] sinx;
  delete [] cosy;
}
```

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение.

Результаты вычисления интеграла должны остаться прежними, а время вычислений должно несколько сократиться по сравнению с предыдущей версией.

Убедитесь, что вывод программы соответствует представленному на рис. 27.

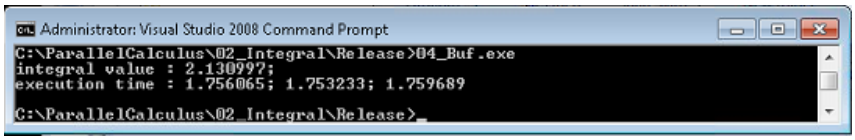

Рис. 27. Результаты последовательной версии с использованием предварительных вычислений и буферизации

Сокращение времени работы вызвано тем, что при вычислении интеграла было задействовано значительно меньше памяти, чем в предыдущей версии.

На графике, представленном на рис. 28, показано сравнение времени выполнения алгоритма с использованием буферизации, а также других реализаций численного интегрирования.

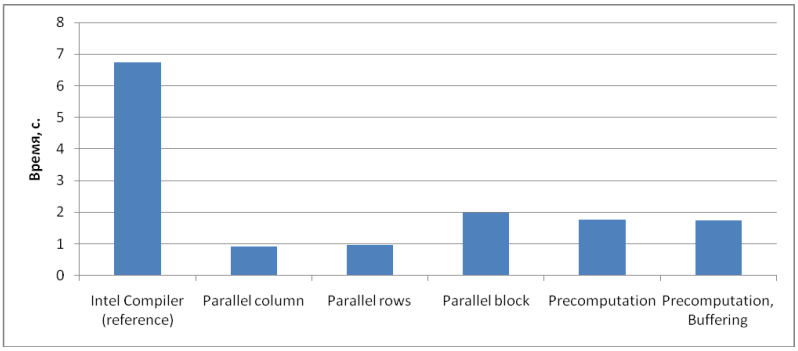

Рис. 28. Сравнение времени численного интегрирования последовательных и параллельных реализаций

3.6. Последовательная версия. Алгоритмическая оптимизация

Предпоследняя оптимизация, которую мы рассмотрим в данной работе, основана на математических особенностях вычисляемого интеграла, а именно на том факте, что функции **`sin`** и **`cos`** периодические.

Внимательно посмотрим на тестовую функцию из формулы (3). Предположим, что количество точек в сетке интегрирования кратно размеру подобранного буфера. В этом случае легко видеть, что итерации цикла по блокам будут вычислять одно и то же значение. Следовательно, значение интеграла можно подсчитать только один раз и умножить его на количество блоков.

Второй момент, на который можно обратить внимание, – часть интеграла можно взять аналитически, сократив тем самым время вычислений (см. формулу (4))

$$\int_{a_1}^{b1}\int_{a2}^{b2}\frac{1}{(b_1-a_1)\cdot(b_2-a_2)}=1\,. \tag{4}$$

Реализуйте алгоритмическое улучшение в проекте **05_AlgOptV2** с использованием техник, рассмотренных в предыдущих разделах.

Часть кода, которая должна получиться, представлена ниже.

```
// количество точек в периоде интегрирования
npi = (int)(2.0 / h);
// правильность размера блока
assert((n1 % npi) == 0);
assert((n2 % npi) == 0);
// вычисление значений sin(x * pi)
for(i = 0; i < npi; i++)
{
  x = a1 + i * h + h / 2;
  sinx[i] = sin(x * PI);
}
// вычисление значений cos(y * pi)
for(j = 0; j < npi; j++)
{
  y = a2 + j * h + h / 2;
  cosy[j] = cos(y * PI);
}
// вычисление интеграла
sum = 0.0;
for(i = 0; i < npi; i++)
{
  for(j = 0; j < npi; j++)
  {
    // вычисление интеграла
    sum += (exp(sinx[i] * cosy[j]) /
           ((b1 - a1) * (b2 - a2))) * h * h;
  }
}

*res = sum * (n1 / npi) * (n2 / npi) + 1;
```

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение.

Результаты вычисления интеграла должны остаться прежними (однако см. соображения выше по поводу порядка вычислений, ограничений разрядной

сетки и вызванных ими возможных проблем), а время вычислений должно кардинально сократиться из-за отсутствия повторяющихся вычислений.

Убедитесь, что вывод программы соответствует представленному на рис. 29.

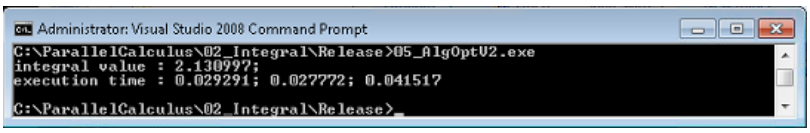

Рис. 29. Результаты последовательной версии с алгоритмической оптимизацией

На графике, представленном на рис. 30, показано сравнение времени выполнения алгоритма с исключенными повторными вычислениями и буферизированной версии последовательного алгоритма.

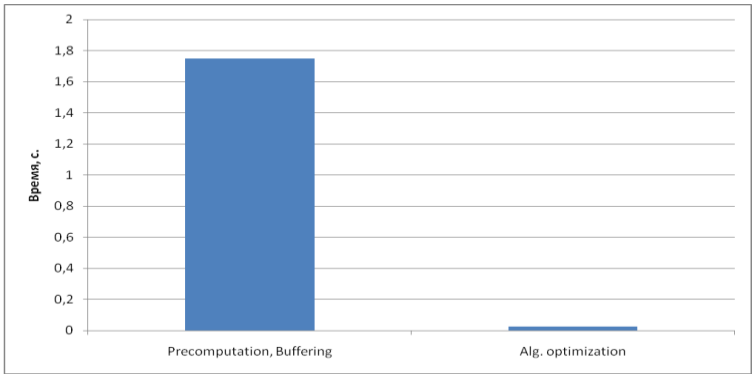

Рис. 30. Сравнение времени численного интегрирования последовательных реализаций

3.7. Параллельная версия. Распараллеливание оптимизированного алгоритма

В качестве последнего шага в данной работе рассмотрим распараллеливание полученной выше оптимизированной реализации. В § 3.3 мы выяснили, что в данной задаче наилучшие результаты дает распараллеливание с разделением сетки интегрирования по столбцам.

Реализуйте параллельную версию алгоритмически улучшенной реализации в проекте **06_AlgOptV2par**.

Вначале объявите необходимые переменные.

```
void integral(const double a1, const double b1,
  const double a2, const double b2, const double h,
  double *res)
{
  int i, j, n1, n2, npi;
```

```
  double sum;// локальная переменная для подсчета интеграла
  double x;  // координата точки сетки по оси x
  double y;  // координата точки сетки по оси y
  double *sinx; // значение sin(x * pi)
  double *cosy; // значение cos(y * pi)
```

Затем подсчитайте размеры сетки интегрирования и выделите необходимую память.

```
  // количество точек сетки интегрирования
  // n1 - по координате x
  // n2 - по координате y
  n1 = (int)((b1 - a1) / h);
  n2 = (int)((b2 - a2) / h);

  // количество точек в периоде интегрирования
  npi = (int)(2.0 / h);
  // правильность размера блока
  assert((n1 % npi) == 0);
  assert((n2 % npi) == 0);

  sinx = new double [npi];
  cosy = new double [npi];
```

Параллельно вычислите необходимые значения функций **sin** и **cos**.

```
  // вычисление значений sin(x * pi)
#pragma omp parallel for private(x)
  for(i = 0; i < npi; i++)
  {
    x = a1 + i * h + h / 2;
    sinx[i] = sin(x * PI);
  }

  // вычисление значений cos(y * pi)
#pragma omp parallel for private(y)
  for(j = 0; j < npi; j++)
  {
    y = a2 + j * h + h / 2;
    cosy[j] = cos(y * PI);
  }
```

Далее распараллельте основные вычислительные циклы. Напомним, что при разделении данных по столбцам необходимо распараллеливать внешний цикл.

```
  // вычисление интеграла
  sum = 0.0;
#pragma omp parallel for private (x, y, j) /
  reduction(+: sum)
  for(i = 0; i < npi; i++)
```

```
  {
    for(j = 0; j < npi; j++)
    {
      // вычисление интеграла
      sum += (exp(sinx[i] * cosy[j]) /
             ((b1 - a1) * (b2 - a2))) * h * h;
    }
  }

  *res = sum * (n1 / npi) * (n2 / npi) + 1;
```

В конце освободите выделенную память.

```
    delete [] sinx;
    delete [] cosy;
}
```

Пересоберите получившийся код (команда **Build→Rebuild Solution**) и запустите его на выполнение.

Убедитесь, что вывод программы соответствует представленному на рис. 31.

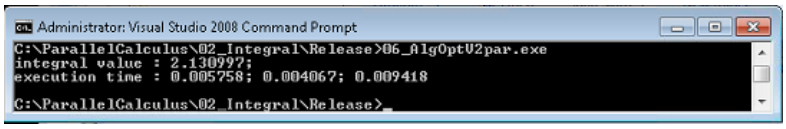

Рис. 31. Результаты параллельной версии с алгоритмической оптимизацией и распараллеливанием

На графике, представленном на рис. 32, показано сравнение времени выполнения алгоритма с исключенными повторными вычислениями с распараллеливанием и без него.

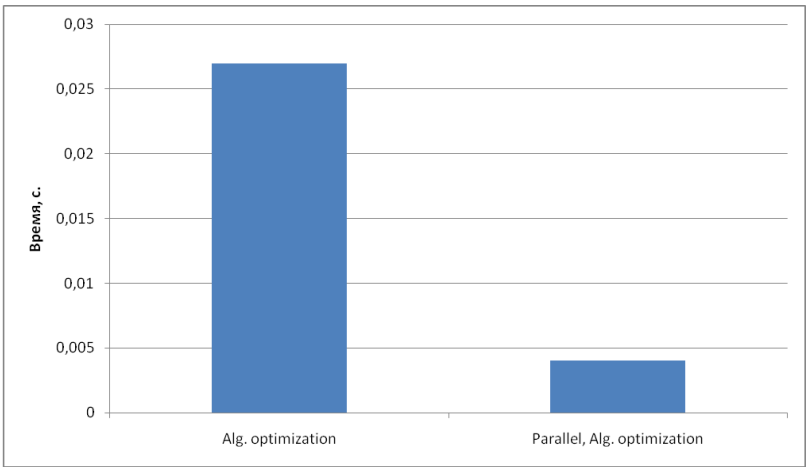

Рис. 32. Сравнение времени численного интегрирования последовательной и параллельной реализаций

4. Дополнительные задания

1. Реализовать алгоритмы, предложенные в параграфах 3.5-3.7 на случай, когда размер буфера не кратен количеству узлов сетки интегрирования.
2. Предложить дальнейший алгоритмический метод сокращения времени вычисления версии, рассмотренной в параграфе 3.6.
3. Рассмотреть возможность использования Intel MKL для векторного вычисления математических функций (функции VML).
4. Рассмотреть другие квадратурные формулы.

5. Литература

1. Ильин В.А., Позняк Э.Г. Основы математического анализа. Часть I. – М.: Физматлит, 2005. – 648 с.
2. Вержбицкий В.М. Численные методы: математический анализ и обыкновенные дифференциальные уравнения. – М.: Высшая школа, 2001. – 382 с.
3. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – Бином. Лаборатория знаний, 2008. – 640 с.
4. Гаврилов В.Р., Иванова Е.Е., Морозова В.Д. Кратные и криволинейные интегралы. Элементы теории поля. – М.: Изд-во МГТУ им. Н.Э.Баумана, 2003. – 493 с.
5. Гергель В.П. Теория и практика параллельных вычислений. – Бином. Лаборатория знаний, 2007. – 424 с.