13

Программирование для нескольких ядер

В ЭТОЙ ГЛАВЕ...

- Основные возможности библиотеки шаблонов для параллельных вычислений
- Как организовать цикл в условиях параллельных вычислений
- Как выполнять несколько задач одновременно
- Как обслужить совместно используемые ресурсы в параллельных операциях
- Как обозначить разделы кода, который не должен обрабатываться несколькими процессорами

Если ваш компьютер или ноутбук относительно новый, то, вероятней всего, его процессор обладает двумя или более ядрами. Если это не так, то пропустите эту главу и переходите к следующей главе. Каждое ядро многоядерного процессора — это независимый процессор, поэтому ваш компьютер может одновременно выполнять код нескольких приложений. Но этим все не ограничивается. Наличие нескольких ядер или процессоров позволяет накладывать вычисления в пределах одного приложения, когда каждый процессор параллельно выполняет отдельную часть вычислений. Поскольку некоторые вычисления вашего приложения способны выполниться параллельно, по крайней мере, оно должно работать быстрее.

В этой главе демонстрируется, как можно использовать несколько ядер в одном приложении. Вы также примените то, что узнали об интерфейсе API Windows в предыдущей главе, с точки зрения параллельных вычислений.

Основы параллельных вычислений

Программирование приложения, использующего несколько ядер или процессоров, требует иной организации вычислений, т.е. не такой, к какой вы привыкли при последовательной обработке. Вычисления следует разделить на ряд задач, которые могут быть выполнены независимо друг от друга, хоть это и не всегда просто или даже возможно. Задачи, которые могут выполниться параллельно, обычно называют *потоками*.

Любая взаимозависимость между потоками накладывает серьезные ограничения на параллельные выполнения. Совместно используемые ресурсы — это наиболее обычное ограничение. Параллельно выполняющиеся задачи, которые обращаются к общим ресурсам, в лучшем случае могут замедлить выполнение (еще медленнее, чем при последовательном выполнении), а в худшем случае это может привести к неправильным результатам или зависанию программы. Простой пример — совместно используемая переменная, значение которой изменяют несколько потоков. Если один поток сохранил результат в переменной, а другой поток изменил его, то результат, сохраненный первым потоком, будет потерян, а результат окажется неправильным. Та же проблема возникает при изменении файлов.

Так каковы же типичные характеристики приложений, которые могут извлечь выгоду из использования нескольких процессоров? Во-первых, эти приложения подразумевают операции, требующие существенного объема процессорного времени, измеряемого в секундах, а не в миллисекундах. Во-вторых, громоздкие вычисления, подразумевающие циклы некоторого вида. В-третьих, операции вычисления могут быть разделены на отдельные, достаточно существенные блоки, которое могут выполняться независимо друг от друга.

Введение в библиотеку шаблонов для параллельных вычислений

Библиотека шаблонов для параллельных вычислений (Parallel Patterns Library — PPL), определенная в файле заголовка `ppl.h`, предоставляет инструментальные средства для реализации параллельной обработки в приложениях, и эти инструменты определены в пространстве имен `Concurrency`. Библиотека PPL определяет три вида средств параллельной обработки.

- Шаблоны для алгоритмов параллельных операций.
- Шаблон класса для обслуживания совместно используемых ресурсов.
- Шаблоны класса для управления и группировки параллельных задач.

Обычно библиотека PPL работает с блоками действий или задачами, определяемыми объектами функции, или, еще чаще, с лямбда-выражениями. Как правило, вам не придется заботиться о создании потоков выполнения или распределении работ по конкретным процессорам, поскольку библиотека PPL позаботится об этом сама. То, что вам действительно предстоит сделать, — так это организовать код в соответствии с требованиями параллельного выполнения.

Алгоритмы параллельной обработки

Библиотека PPL предоставляет три алгоритма инициализации параллельной обработки на нескольких ядрах.

- Алгоритм `parallel_for` — эквивалент цикла `for`, выполняющий итерации цикла параллельно.
- Алгоритм `parallel_for_each` параллельно выполняет повторяющиеся операции с контейнером STL.
- Алгоритм `parallel_invoke` параллельно выполняет набор из двух или нескольких независимых задач.

Алгоритмы определены как шаблоны, но я не буду вникать в подробности определения самих шаблонов. По желанию можете обратить внимание на файл заголовка `ppl.h`. Давайте лучше рассмотрим, как использовать каждый из алгоритмов.

Использование алгоритма `parallel_for`

Алгоритм `parallel_for` выполняет параллельные итерации цикла для одной задачи, определенной объектом функции или лямбда-выражением. Итерации цикла контролируются целочисленным индексом, значение которого увеличивается при каждой итерации, пока не будет достигнут указанный предел, как и у обычного цикла. У объекта функции или лямбда-выражения, выполняющего задачу, должен быть один параметр; при каждой итерации текущее значение индекса будет передано как аргумент, соответствующий этому параметру. Из-за параллельного характера операции итерации не будут выполняться ни в каком специфическом порядке.

Есть две версии алгоритма `parallel_for`. Форма первого приведена ниже.

```
parallel_for(начальное_значение, конечное_значение, объект_функции);
```

Этот код выполняет задачу, определенную третьим аргументом, со значениями индекса от *начальное_значение* до *конечное_значение*`-1` и стандартным приращением 1. Первые два аргумента должны иметь одинаковый целочисленный тип. В противном случае при компиляции произойдет ошибка. Как уже упоминалось, поскольку итерации выполняются параллельно, они не будут осуществляться последовательно. Таким образом, каждое выполнение объекта функции или лямбда-выражения будет независимым от других, и результат не должен зависеть от решения задачи в определенной последовательности.

Ниже приведен фрагмент кода, демонстрирующий работу алгоритма `parallel_for`.

```
const size_t count(100000);
long long squares[count];
Concurrency::parallel_for(static_cast<size_t>(0), count,
                          [&squares](size_t n)
                          { squares[n] = (n+1)*(n+1); });
```

Вычислительная задача для каждой итерации цикла определяется лямбда-выражением, являющимся третьим аргументом. Директива фиксации обращается к массиву квадратов по ссылке. Функция вычисляет квадрат *n*`+1`, где *n* — текущее значение индекса, передаваемое лямбда-выражению, а результат сохраняется в энном элементе массива `squares`. Выполнение будет происходить для значений *n*, от `0` до `count-1`, как определено первыми двумя аргументами. Обратите внимание на приведение первого аргумента. Поскольку аргумент `count` имеет тип `size_t`, код без этого не будет компилироваться. Поскольку `0` — это литерал типа `int`, без приведения компилятор не сможет решить, использовать ли ему тип `int` или тип `size_t` как параметр типа для шаблона алгоритма `parallel_for`.

Хотя этот фрагмент кода демонстрирует возможность применения алгоритма parallel_for, увеличение производительности в данном случае будет невелико, а в действительности может оказаться даже ниже, чем при последовательном выполнении. Здесь очень мало вычислений на каждой итерации, но неизбежно появляются некоторые дополнительные затраты на распределение задач по процессорам при каждой итерации. Дополнительные затраты могут превзойти экономию времени при выполнении вычислений, которых относительно немного. Не стоит также обольщаться этим простым примером и полагать, что реализация параллельной обработки проста и что можно просто заменить цикл for алгоритмом parallel_for. Как вы увидите далее в этой главе, в реальной ситуации может возникнуть множество сложностей всех видов.

Хотя индекс в алгоритме parallel_for всегда должен быть указан, в выполняемой функции можно задать декремент значений индексов.

```
const size_t count(100000);
long long squares[count];
Concurrency::parallel_for(static_cast<size_t>(0), count,
                          [&squares, &count](size_t n)
                          { squares[count-n-1] = (n+1)*(n+1); });
```

Здесь квадраты значений индекса, переданные лямбда-выражению, будут сохраняться в массиве в порядке убывания.

Вторая версия алгоритма parallel_for имеет следующий вид.

```
parallel_for(начальное_значение, конечное_значение, приращение, указатель_на_функцию);
```

Второй аргумент определяет *приращение*, используемое при выполнении итераций от *начальное_значение* до *конечное_значение*-1, таким образом, здесь предусмотрен инкремент, отличный от 1. Второй аргумент должен быть положительным, поскольку в алгоритме parallel_for допустимы только увеличивающиеся значения индексов. Все три первых аргумента должны иметь одинаковый целочисленный тип. Ниже приведен фрагмент кода, использующий эту форму алгоритма parallel_for.

```
size_t start(1), end(300001), step(3);
vector<double> cubes(end/step + 1);
Concurrency::parallel_for(start, end, step, [&cubes](int n)
                      { cubes[(n-1)/3] = static_cast<double>(n)*n*n; });
```

Этот фрагмент кода вычисляет кубы целых чисел от 1 до 300001 с шагом 3 и сохраняет значения в векторе cubes. Таким образом, будут сохранены кубы чисел 1, 4, 7, 10 и до 29998. Поскольку это параллельная операция, значения в векторе могут сохраняться не в такой последовательности.

В связи с тем что операции с контейнерами STL не потокобезопасны, при использовании их в параллельных операциях следует соблюдать большую осторожность. Здесь это необходимо, чтобы заранее зарезервировать достаточно пространства в векторе и сохранить результаты операции. Если бы для сохранения значений использовалась функция push_back(), код не сработал бы.

Поскольку алгоритм parallel_for — это шаблон, необходимо позаботиться об одинаковых типах для аргументов *начальное_значение*, *конечное_значение* и *приращение*. Это очень просто упустить, когда литералы смешиваются с переменными. Например, код, приведенный ниже, не будет откомпилирован.

```
size_t end(300001);
vector<double> cubes(end/3 + 1);
Concurrency::parallel_for(1, end, 3, [&cubes](size_t n)
                      { cubes.push_back (static_cast<double>(n)*n*n); });
```

Он не компилируется потому, что первый и третий аргументы являются литералами и имеют тип `int`, а счетчик имеет тип `size_t`. В результате компилятору неизвестен тип экземпляра шаблона.

Использование алгоритма `parallel_for_each`

Алгоритм `parallel_for_each` подобен алгоритму `parallel_for`, но функция работает в контейнере STL. Количество выполняемых итераций определяется парой итераторов. Этот алгоритм имеет следующую форму.

`начальный_итератор конечный_итератор объект_функции`

```
Concurrency::parallel_for_each(начальный_итератор, конечный_итератор,
                               объект_функции);
```

Объект функции, определенный третьим аргументом, должен иметь один параметр типа, хранящегося в контейнере, к которому относятся итераторы. Это позволит получить доступ к объекту в контейнере, заданному текущим итератором. Перебор объектов осуществляется в диапазоне от *`начальный_итератор`* до *`конечный_итератор`*`-1`.

Ниже приведен фрагмент кода, иллюстрирующий, как это работает.

```
std::array<string, 6> people = {"Mel Gibson", "John Wayne",
                                "Charles Chaplin", "Clint Eastwood",
                                "Jack Nicholson", "George Clooney"};
Concurrency::parallel_for_each(people.begin(), people.end(),
                               [](string& person)
{
    size_t space(person.find(' '));
    std::string first(person.substr(0, space));
    std::string second(person.substr(space + 1, person.length()
                                     - space - 1));
    person = second + ' ' + first;
});

std::for_each(people.begin(), people.end(), [](std::string& person)
              {std::cout << person << std::endl; });
```

Алгоритм `parallel_for_each` работает с массивом `people`. Безусловно, у реального приложения было бы больше шести элементов. Третий аргумент определяет функцию, осуществляющую работу при каждой итерации. Параметр — ссылка на тип `string`, поскольку элементы массива `people` имеют тип `string`. Функция меняет порядок имен в каждой строке на обратный. Последний оператор использует алгоритм библиотеки STL `for_each` для вывода содержимого массива `people`. Как можно заметить, у алгоритма библиотеки PPL `parallel_for_each`, по существу, тот же самый синтаксис, что и у алгоритма библиотеки STL `for_each`.

Обратите внимание на то, что ни один из параллельных алгоритмов не предусматривает преждевременного завершения работы. Для цикла `for`, например, можете использовать оператор `break`, завершающий цикл, но у параллельных алгоритмов такого нет. Единственный способ завершить работу параллельного алгоритма до завершения всех параллельных операций — это передать исключение в функции, определяющей выполняемые действия. Это немедленно остановит все параллельные операции. Вот, например, как это можно сделать.

```
int findIndex(const std::array<string, 6>& people,
              const std::string& name)
```

```
{
    try
    {
        Concurrency::parallel_for(static_cast<size_t>(0), people.size(),
                                  [=](size_t n)
        {
            size_t space(people[n].find(' '));
            if(people[n].substr(0, space) == name)
                throw n;
        });
    } catch(size_t index) { return index; }
    return -1;
}
```

Эта функция ищет в массиве из предыдущего фрагмента специфическую строку имен. Алгоритм `parallel_for` сравнивает имя до первого пробела в текущем элементе массива `people` со вторым аргументом функции `findIndex()`. Если они совпадают, лямбда-выражение передает исключение, содержащее текущий индекс массива `people`. Оно будет обработано в блоке `catch` функции `findIndex()`, а переданное значение будет возвращено. Передача исключения завершает все текущие действия алгоритма `parallel_for`. Функция `findIndex()` возвращает значение `-1`, если операция алгоритма `parallel_for` заканчивается без передачи исключения, поскольку это означает, что имя не было найдено. Посмотрим, работает ли это.

Практическое занятие

Завершение работы алгоритма `parallel_for`

Чтобы опробовать функцию `findIndex()`, необходимо добавить некоторый код к предыдущему фрагменту, который использует алгоритм `parallel_for_each` для изменения порядка имен на обратный. Ниже приведен полный код программы.


```
// Ex13_01.cpp Завершение параллельной операции
#include <iostream>
#include "ppl.h"
#include <array>
#include <string>
using namespace std;
using namespace Concurrency;

// Найти имя как первое имя в каждом элементе
int findIndex(const array<std::string, 6>& people, const string& name)
{
    try
    {
        parallel_for(static_cast<size_t>(0), people.size(), [=](size_t n)
        {
            size_t space(people[n].find(' '));
            if(people[n].substr(0, space) == name)
                throw n;
        });
    } catch(size_t index) { return index; }
    return -1;
}
```

```
int main()
{
    array<string, 6> people = {"Mel Gibson", "John Wayne",
                               "Charles Chaplin", "Clint Eastwood",
                               "Jack Nicholson", "George Clooney"};
    parallel_for_each(people.begin(), people.end(), [](string& person)
    {
        size_t space(person.find(' '));
        string first(person.substr(0, space));
        string second(person.substr(space+1, person.length()-space-1));
        person = second +' ' + first;
    });
    for_each(people.begin(), people.end(), [](string& person)
             { cout << person << endl; });
    string name("Eastwood");
    size_t index(findIndex(people, name));
    if(index == -1)
        cout << name << " not found." << endl;
    else
        cout << name << " found at index position " << index << '.'
             << endl;
    return 0;
}
```

Фрагмент кода Ex6_01.cpp

Вывод этого примера показан ниже.

```
Gibson Mel
Wayne John
Chaplin Charles
Eastwood Clint
Nicholson Jack
Clooney George
Eastwood found at index position 3.
```

Как это работает

Алгоритм `parallel_for_each` меняет местами имена и фамилии в каждом элементе массива `people`, используя параллельные операции, описанные в предыдущем разделе. Вывод, созданный алгоритм STL `for_each`, демонстрирует, что все работает так, как надо.

Затем вы создаете искомую строку `name`. Переменная `index` инициализируется значением, возвращенным функцией `findIndex()`. Оператор `if` выводит сообщение в зависимости от значения переменной `index`, и последняя строка вывода демонстрирует, что значение переменной `name` действительно было найдено в массиве `people`.

Использование алгоритма `parallel_invoke`

Алгоритм `parallel_invoke` определяется шаблоном, который позволяет решать параллельно до десяти задач. Вполне очевидно, что нет никакого смысла пытаться решать параллельно больше задач, чем есть процессоров на вашем компьютере. Вы определяете каждую задачу, которая будет решаться объектом функции, который, конечно, может быть лямбда-выражением. Таким образом, аргументы алгоритма `parallel_invoke` — это объекты функции, определяющие задачи, которые будут решаться параллельно. Рассмотрим пример.

```
std::array<double, 500> squares;
std::array<double, 500> values;
Concurrency::parallel_invoke(
    [&squares]
    { for(size_t i = 0 ; i<squares.size() ; ++i)
        squares[i] = (i+1)*(i+1);
    },
     [&values]
    { for(size_t i = 0 ; i<values.size() ; ++i)
        values[i] = std::sqrt(static_cast<double>((i+1)*(i+1)*(i+1)));
});
```

Аргументами алгоритма `parallel_invoke` являются два разных лямбда-выражения. Первое сохраняет квадраты последовательных целых чисел в массиве `squares`, а второе сохраняет в массиве `values` квадратный корень куба последовательных целых чисел. Эти лямбда-выражения полностью независимы друг от друга, а потому могут быть выполнены параллельно.

Приведенные здесь примеры и фрагменты кода не были теми вычислениями, которые могли бы извлечь реальную пользу от параллельного выполнения. Давайте рассмотрим вычисления, которые могут извлечь реальную пользу из параллельного выполнения.

Реальная проблема параллельной обработки

Чтобы исследовать практичность параллельной обработки и увидеть ее преимущества, необходимо вычисление, которое, являясь достаточно коротким, гарантирует выгоду параллельной обработки. Понадобится также нечто, что можно использовать для исследования некоторых проблем, которые возникают при параллельных операциях.

Проблема, которую я выбрал, не требует большого количества кода, но подразумевает выполнение реальных вычислений с *комплексными числами*. Если вы никогда не слышали о комплексных числах или, возможно, слышали, но забыли, я предварительно объясню, что они собой представляют и как с ними работать. Затем мы перейдем к коду. Честное слово: это не трудно, а результаты интересны.

Комплексными называют числа в формате *a + bi*, где *i* — квадратный корень –1. Таким образом, `1 + 2i` и `0.5 - 1.2i` — это примеры комплексных чисел. Часть *a* называется вещественной частью комплексного числа, а *b*, умноженное на *i*, называется мнимой частью. Комплексные числа зачастую отображаются на комплексной плоскости, как показано на рис. 13.1.

При сложении двух комплексных чисел вещественные и мнимые части суммируются отдельно. Таким образом, сумма чисел *1 + 2i* и *0.5 - 1.2i* равна *(1+ 0.5) + (2 - 1.2)i*, следовательно, результат будет *1.5 + 0.8i*.

Как осуществляется умножение двух комплексных чисел *a + bi* и *c + di*, показано ниже.

```
a, c + di. + bi(c + di)
```

Не забывайте, что *i* — это квадратный корень –1, поэтому i^2 равно –1, и выражение можно переписать так.

```
ac + adi + bci - bd
```

В результате получается комплексное число

```
(ac - bd) + (ad + bc)i
```

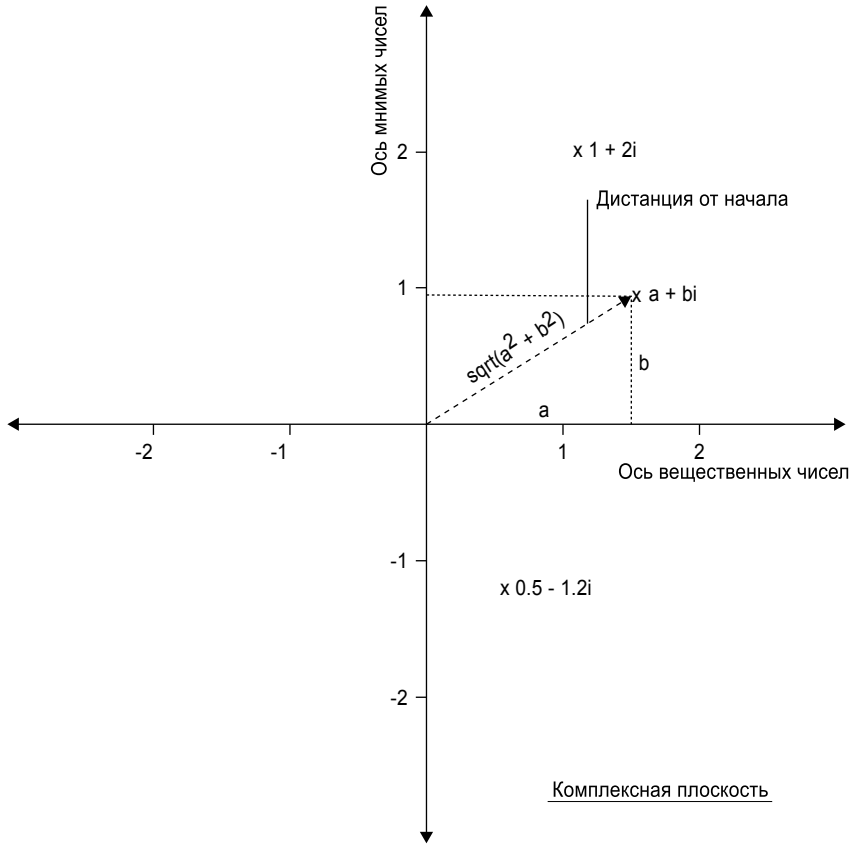

Рис. 13.1. Комплексная плоскость

Умножение двух чисел из примера выше приведет к следующему: `(0.5 + 2.4) + (-1.2 + 1.0)i`, что составляет `2.9 - 0.2i`. Умножение числа `a + bi` на само себя выглядит так.

```
(a2 - b2) + 2abi
```

Теперь выясним, в чем проблема. Набор Мандельброта — очень интересный раздел комплексной плоскости, который определяется при итерации уравнения

$$Z_{n+1} = Z^2_n + c$$

где $Z_0 = c$ для всех точек c в области комплексной плоскости. При выполнении итераций я подразумеваю, что для любой точки c в комплексной плоскости вы приравниваете Z_0 к c и используете уравнение c^2+c для вычисления Z_1. Потом значение Z_1 используется для вычисления значения Z_2, с использованием того же уравнения, затем по значению Z_2 вычисляется значение Z_3 и т.д. Так что же за точки находятся в наборе Мандельброта? Если дистанция от начальной точки (см. рис. 13.1) какой-нибудь Z_n превысит 2, то продолжение итераций ведет к точке далее и дальше от исходной точки до бесконечности, поэтому дальнейшие итерации не должны выполняться. Все удаляющиеся точки *не* входят в набор Мандельброта. Любая точка, которая не удаляется таким образом после достаточно большого количества итераций, принадлежит набору

Мандельброта. Любая точка, начинающая с дистанции больше 2 от начальной точки, не будет принадлежать набору Мандельброта. Таким образом, это даст нам некоторое общее представление об области в комплексной плоскости и ее виде.

Ниже приведена функция, осуществляющая итерации уравнения для набора Мандельброта.

```
size_t IteratePoint(double zReal, double zImaginary, double cReal,
                    double cImaginary)
{
    double zReal2(0.0), zImaginary2(0.0);  // компоненты квадрата z
    size_t n(0);
    for(; n < MaxIterations ; ++n) // Итерация уравнения Zn = Zn-1**2 + K
    {                              // для набора Мандельброта K = c и Z0 = c
        zReal2 = zReal*zReal;
        zImaginary2 = zImaginary*zImaginary;
        if(zReal2 + zImaginary2 > 4) // Если дистанция от начала > 2,
            break; // точка уйдет в бесконечность, на этом закончить

        // Вычислить следующее значение Z
        zImaginary = 2*zReal*zImaginary + cImaginary;
        zReal = zReal2 - zImaginary2 + cReal;
    }
    return n;
}
```

Функция `IteratePoint()` выполняет итерации уравнения $Z_{n+1}=Z^2_n+C$ с отправной точкой Z, определенной первыми двумя параметрами и комплексной константой C, определенной последними двумя параметрами. Значение `MaxIterations` — это максимальное количество вводов точки Z в уравнение, данная константа должна быть определена как глобальная. Цикл `for` вычисляет новую точку Z, и если ее дистанция от исходной точки превышает 2, то цикл заканчивается, возвращается и текущее значение переменной `n`. Выражение оператора `if` фактически сравнивает квадрат дистанции от исходной точки со значением 4, избегая довольно дорогостоящего вычисления квадратного корня. Если цикл завершится нормально, то переменная `n` будет содержать значение константы `MaxIterations`, которое и будет возвращено.

Так что, цикл `for` в функции `IteratePoint()` является кандидатом на параллельные операции? Боюсь, что нет. Каждая итерация цикла требует доступа к точке Z от предыдущей итерации, таким образом, это, по существу, последовательная операция. Но есть и другие возможности. Чтобы выяснить экстент набора Мандельброта, необходимо вызвать эту функцию для каждой точки в области комплексной плоскости. В том контексте появляется большой потенциал для параллельного вычисления.

Чтобы визуализировать набор Мандельброта, отобразим его в клиентской области окна; пиксели его клиентской области окна будем рассматривать как точки комплексной плоскости. Таким образом, мы нуждаемся в цикле, который проверяет каждый пиксель в клиентской области на предмет принадлежности к набору Мандельброта. В общих чертах цикл будет выглядеть так, как показано ниже.

```
for(int y = 0 ; y < imageHeight ; ++y) // Перебрать строки изображения
{
    for(int x = 0 ; x < imageWidth ; ++x) // Перебрать пиксели в строке
    // Итерация текущей точки x+yi...
}
// Отобразить строку номер y...
```

Каждый пиксель изображения будет соответствовать конкретной точке в области комплексной плоскости. Координаты пикселей располагаются от 0 до `imageWidth` вдоль горизонтальной линии и от 0 до `imageHeight` вниз вдоль оси Y клиентской области. Необходимо соотнести координаты пикселей с интересуемой областью комплексной плоскости. Хорошей областью для исследования набора Мандельброта является окно, содержащее вещественные части вдоль оси X от –2,1 до +2,1 и мнимые части вдоль оси Y от –1,3 до +1,3. Однако клиентская область окна может и не иметь такого же соотношения, как в идеале, поэтому следует позаботиться о наложении пикселей на комплексную плоскость хотя бы так, чтобы обеспечить одинаковый масштаб по обеим осям. Это предотвратит растяжение или сжатие формы набора Мандельброта. Для этого можем вычислить коэффициенты преобразования из координат пикселей в координаты комплексной плоскости.

```
// Оси клиентской области
const double realMin(-2.1); // Минимальное вещественное значение
double imaginaryMin(-1.3);  // Минимальное мнимое значение
double imaginaryMax(+1.3);  // Максимальное мнимое значение

// Установить максимум вещественного значения, чтобы оси имели
// одинаковый масштаб
double realMax(realMin+(imaginaryMax-
                        imaginaryMin)*imageWidth/imageHeight);

// Получить коэффициенты масштабирования для координат пикселей
double realScale((realMax-realMin)/(imageWidth-1));
double imaginaryScale((imaginaryMax-imaginaryMin)/(imageHeight-1));
```

Значение переменной `realMax` — это максимальное вещественное значение, вычисляемое так, чтобы у осей вещественных и мнимых частей был одинаковый масштаб. Переменные `realScale` и `imaginaryScale` содержат коэффициенты, которые мы можем использовать для умножения координат `x` и `y` пикселей, преобразуя их в соответствующие значения комплексной плоскости.

А теперь соберем все фрагменты в рабочую программу.

Практическое занятие Отображение набора Мандельброта

Напишем программу API Windows. Создайте новый проект приложения Win32 со стандартными параметрами Windows. Я назвал его `Ex13_02A`. Когда проект будет создан, измените значение его свойства `Character Set` на `Not Set`, поскольку мы не будем использовать символы Unicode. Вместо настроек окна, заданных по умолчанию, измените вызов функции `CreateWindow()` в функции `InitInstance()`.

```
hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
                    100, 100, 800, 600, NULL, NULL, hInstance, NULL);
```

Теперь окно будет расположено в позиции 100, 100, а его размер составит 800×600 пикселей. Если ваш экран имеет существенно больше пикселей, изменяйте размер окна так, чтобы его было видно. Но помните, что чем больше пикселей находится в окне, тем дольше будет выполняться программа.

В этом примере и других примерах этой главы я использую для нулевого указателя значение `NULL` исключительно для совпадения с кодом, который создается автоматически. В других местах книги я использую новое ключевое слово языка `nullptr`.

Теперь добавьте следующее определение в глобальную область видимости.

```
const size_t MaxIterations(8000); // Максимум итерации до бесконечности
```

Значение константы `MaxIterations` зависит от точности представления комплексной плоскости. Чем больше подробностей вы хотите отобразить, тем выше должно быть это значение. Установленное здесь значение намного выше, чем необходимо для типичной клиентской области, но я сделал это для того, чтобы время вычисления набора Мандельброта заняло значительное количество секунд.

Затем добавим три глобальных прототипа функций после существующих прототипов в начале файла исходного кода.

```
size_t IteratePoint(double zReal, double zImaginary, double cReal,
                    double cImaginary);
COLORREF Color(int n);      // Выбор цвета пикселя на основании n
void DrawSet(HWND hwnd);    // Нарисовать набор Мандельброта
```

Теперь можно добавить определение функции `IteratePoint()` в конец файла исходного кода, как и в предыдущем разделе.

Добавьте следующее определение функции `Color()`.


```
COLORREF Color(int n)
{
    if(n == MaxIterations) return RGB(0, 0, 0);

    const int nColors = 16;
    switch(n%nColors)
    {
        case 0:  return RGB(100,100,100);
        case 1:  return RGB(100,0,0);
        case 2:  return RGB(200,0,0);
        case 3:  return RGB(100,100,0);
        case 4:  return RGB(200,100,0);
        case 5:  return RGB(200,200,0);
        case 6:  return RGB(0,200,0);
        case 7:  return RGB(0,100,100);
        case 8:  return RGB(0,200,100);
        case 9:  return RGB(0,100,200);
        case 10: return RGB(0,200,200);
        case 11: return RGB(0,0,200);
        case 12: return RGB(100,0,100);
        case 13: return RGB(200,0,100);
        case 14: return RGB(100,0,200);
        case 15: return RGB(200,0,200);
        default: return RGB(200,200,200);
    };
}
```

Фрагмент кода Ex13_02A.cpp

Эта функция выбирает цвет пикселей на основании значения параметра `n`, возвращаемого функцией `IteratePoint()`. Макрокоманда `RGB` формирует значение `DWORD`, определяющее цвет из трех 8-битовых значений для интенсивности основных цветов: красного, зеленого и синего. Каждый компонент цвета может иметь значение от 0 до 255. Все нулевые значения соответствуют черному цвету, а все значения 255 — белому.

Процесс выбора цвета на основании значения параметра n произвольный. Я решил использовать остаток после деления n на количество цветов, поскольку это хорошо и разумно согласуется со значением, которое я установил для константы MaxIterations. Более популярен подход выбора цвета на основании попадания значения параметра n в один из интервалов значений от 0 до MaxIterations-1, где количество интервалов соответствует количеству цветов, а черному цвету соответствует значение константы MaxIterations. Но это работает лучше с более низким значением константы MaxIterations. Вы можете попробовать самостоятельно изменить механизм выбора цвета пикселя.

Можете также добавить следующее определение функции DrawSet() в файл исходного кода.


Доступно для загрузки на Wrox.com

```
void DrawSet(HWND hWnd)
{
    // Получить размерности клиентской области, составляющие размер
    // изображения
    RECT rect;
    GetClientRect(hWnd, &rect);
    int imageHeight(rect.bottom);
    int imageWidth(rect.right);

    // Создать один ряд пикселей изображения
    HDC hdc(GetDC(hWnd));                // Получить контекст устройства
    HDC memDC = CreateCompatibleDC(hdc); // Получить контекст устройства
                                         // для рисования пикселей
    HBITMAP bmp = CreateCompatibleBitmap(hdc, imageWidth, 1);
    HGDIOBJ oldBmp = SelectObject(memDC, bmp); // Выбрать изображение
                                               // в DC
    // Оси клиентской области
    const double realMin(-2.1); // Минимальное вещественное значение
    double imaginaryMin(-1.3);  // Минимальное мнимое значение
    double imaginaryMax(+1.3);  // Максимальное мнимое значение

    // Задать максимум вещественных, чтобы оси имели одинаковый масштаб
    double realMax(realMin+(imaginaryMax-
                            imaginaryMin)*imageWidth/imageHeight);

    // Получить коэффициенты масштабирования для координат пикселей
    double realScale((realMax-realMin)/(imageWidth-1));
    double imaginaryScale((imaginaryMax-imaginaryMin)/(imageHeight-1));
    double cReal(0.0), cImaginary(0.0); // Сохранить компоненты c
    double zReal(0.0), zImaginary(0.0); // Сохранить компоненты z
    // Перебрать строки изображения
    for(int y = 0 ; y < imageHeight ; ++y)
    {
        zImaginary = cImaginary = imaginaryMax - y*imaginaryScale;
        // Перебрать пиксели в строке
        for(int x = 0 ; x < imageWidth ; ++x)
        {
            zReal = cReal = realMin + x*realScale;

            // Установить цвет пикселя на основании n
            SetPixel(memDC, x, 0, Color(IteratePoint(zReal, zImaginary,
                                                     cReal, cImaginary)));
        }
```

```
            // Передать строку пикселей в контекст устройства клиентской
            // области
            BitBlt(hdc, 0, y, imageWidth, 1, memDC, 0, 0, SRCCOPY);
        }
        SelectObject(memDC, oldBmp);
        DeleteObject(bmp);     // Удалить изображение
        DeleteDC(memDC);       // and our working DC и рабочий DC
        ReleaseDC(hWnd, hdc); // and client area DC Освободить клиентскую
    область DC
    }
```

Фрагмент кода Ex13_02A.cpp

Эта версия функции выполняется строго последовательно. Мы дойдем до параллельной версии, проделав соответствующую работу. Параметр типа `HWND` предоставляет доступ к текущему окну в пределах функции. Это необходимо для выяснения размера клиентской области и рисования в ней. Вызов функции `GetClientRect()` интерфейса API Windows выдает данные о клиентской области окна, которое вы определяете первым аргументом, в структуре `RECT`, которую вы предоставляете вторым аргументом. Структура `RECT` будет содержать координаты левых верхних и правых нижних точек клиентской области окна. Открытыми членами структуры, содержащими координаты x, y верхнего левого угла, являются `left` и `top`, а содержащими координаты правого нижнего угла — `right` и `bottom`. Координатами верхней левой точки будет 0,0, таким образом, ширина и высота клиентской области будут определены значениями переменных-членов `right` и `bottom`.

Для рисования в клиентской области необходим *контекст устройства* (Device Context — DC) — структура, используемая для рисования или отображения текста в окне. Вы получаете контекст устройства от операционной системы Windows в результате вызова функции `GetDC()` при передаче как аргумента дескриптора окна, с которым предстоит работать. Функция возвращает дескриптор контекста устройства, инкапсулирующего клиентскую область окна. Мы не будем записывать пиксели непосредственно в данный контекст устройства, а соберем их в строку изображения и только затем передадим ее в контекст устройства. Для этого необходимо сначала создать в памяти совместимый контекст устройства, `memDC`, обладающий тем же диапазоном цветов пикселей, что и у клиентской области окна контекст устройства. Первоначально этот контекст устройства будет в состоянии хранить область только 1×1 пиксель, поэтому его необходимо откорректировать. Затем вызов функции `CreateCompatibleBitmap()` создает дескриптор растрового объекта `bmp`, способного хранить один ряд пикселей (высота `imageWidth` размером в 1 пиксель). Вызов функции `SelectObject()` заменяет существующий объект в контексте устройства, указанный первым аргументом, тем, который был передан вторым аргументом. Так, выбор объекта `bmp` в контекст устройства `memDC` обеспечивает ему емкость, достаточную для содержания строки пикселей изображения.

После создания и инициализации переменных, содержащих точку *Z* и константу `c`, следуют два вложенных цикла `for`. Внешний цикл перебирает строки изображения, а внутренний перебирает пиксели в строке.

Внутренний цикл собирает строку пикселей изображения в контексте `memDC`, используя функцию API `SetPixel()` для установки цвета каждого пикселя, определенного значением, возвращенным функцией `IteratePoint()`. Аргументами функции `SetPixel()` являются контекст устройства, а также координаты *x* и *y* пикселя, цвет которого устанавливается.

Контекст memDC многократно используется для каждого ряда пикселей в клиентской области, таким образом, прежде чем перейти к следующей строке, необходимо записать содержимое контекста memDC в контекст устройства клиентской области hdc. Для этого используется функция BitBlt() (*Bit Bl*ock *T*ransfer — передача блока битов). Аргументами этой функции являются: контекст устройства назначения (hdc); координаты первого пикселя; количество пикселей в линии (imageWidth) и количество линий (1); исходный контекст устройства (memDC); координаты *x* и *y* первого пикселя исходного контекста устройства (0,0) и код растровой операции (SRCCOPY), определяющий, как исходные пиксели должны быть объединены с пикселями назначения.

И наконец, по завершении вложенных циклов удаляем растровое изображение и ресурс контекста устройства, а также освобождаем контекст устройства, полученный от операционной системы Windows, поскольку они больше не нужны.

Последний фрагмента кода, который необходимо добавить, находится в функции WndProc().

```
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Здесь добавить весь код рисования...
    DrawSet(hWnd);
    EndPaint(hWnd, &ps);
    break;
```

Осталось добавить вызов функции DrawSet() как реакцию на сообщение WM_PAINT.

После компиляции и запуска примера его окно должно выглядеть так, как показано на рис. 13.2.

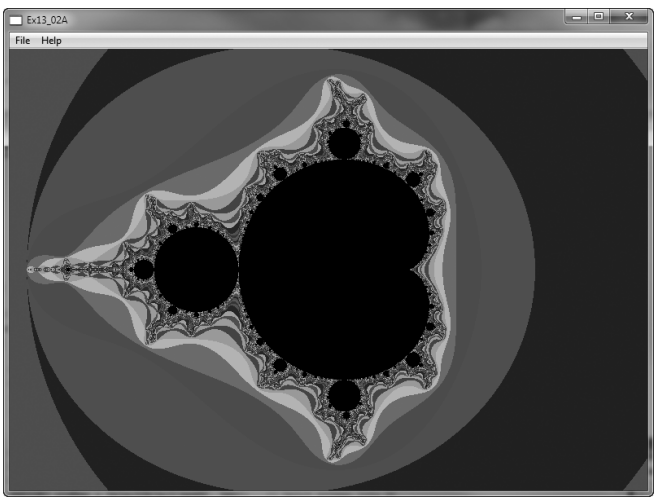

Рис. 13.2. Пример после запуска

Конечно, на вашем экране это будет выглядеть куда ярче. Набор Мандельброта — это черная область. Безусловно, набор имеет конечную область, поскольку он находится в круге радиусом 2, но его периметр имеет бесконечную длину. Кроме того, любой сегмент его периметра, независимо от величины, также имеет бесконечную длину, таким образом, существуют неограниченные возможности для исследования границ набора при масштабировании изображения. Цветные области вокруг набора создают весьма интересные узоры, особенно при масштабировании.

Практическое занятие

Рисование набора Мандельброта с помощью алгоритма `parallel_invoke`

Предстоит создать новую версию функции `DrawSet()` по имени `DrawSetParallel()`, но не заменять исходную. Сохраните обе версии в исходном коде примера. Они будут использованы в последующей версии этой программы. Эта версия программы (`Ex13_02B`) есть в загружаемом коде.

Для повышения производительности программы мы могли бы использовать алгоритм `parallel_invoke`, чтобы вычислять две или более линии изображения одновременно, в зависимости от количества ядер или процессоров компьютера. Но главным ограничением является тот факт, что вы не должны параллельно писать в контекст устройства, поскольку контексты устройства — это строго последовательные средства. Это означает, что вызовы функций `SetPixel()` и `BitBlt()` являются проблемой.

Один из способов справиться с вызовом функции `SetPixel()` подразумевает создание в памяти отдельного контекста устройства и растрового изображения для каждого параллельного вычисления строки. Если вы предоставляете отдельные контексты устройства и растровые изображения для параллельных задач, которые передаете алгоритму `parallel_invoke`, то они могут выполняться, не мешая друг другу. Я определю функцию `DrawSetParallel()` для двух ядер, поскольку это все, что я имею на своем компьютере. Если вы имеете больше, то дополните код соответственно. Код параллельной версии функции `DrawSet()` мог бы выглядеть так, как показано ниже.

```
void DrawSetParallel(HWND hWnd)
{
    HDC hdc(GetDC(hWnd)); // Получить контекст устройства

    // Получить размерности клиентской области, составляющие размер
    // изображения
    RECT rect;
    GetClientRect(hWnd, &rect);
    int imageHeight(rect.bottom);
    int imageWidth(rect.right);

    // Создать один ряд пикселей изображения
    HDC memDC1 = CreateCompatibleDC(hdc); // Получить контекст устройства
                                          // для рисования пикселей
    HDC memDC2 = CreateCompatibleDC(hdc); // Получить контекст устройства
                                          // для рисования пикселей
    HBITMAP bmp1 = CreateCompatibleBitmap(hdc, imageWidth, 1);
    HBITMAP bmp2 = CreateCompatibleBitmap(hdc, imageWidth, 1);
    HGDIOBJ oldBmp1 = SelectObject(memDC1, bmp1); // Выбрать изображение
                                                  // в DC1
    HGDIOBJ oldBmp2 = SelectObject(memDC2, bmp2); // Выбрать изображение
                                                  // в DC2
    // Оси клиентской области
    const double realMin(-2.1); // Минимальное вещественное значение
    double imaginaryMin(-1.3);  // Минимальное мнимое значение
    double imaginaryMax(+1.3);  // Максимальное мнимое значение

    // Установить максимум вещественного значения, чтобы оси имели
    // одинаковый масштаб
    double realMax(realMin+(imaginaryMax-
```

```
                        imaginaryMin)*imageWidth/imageHeight);

    // Получить коэффициенты масштабирования для координат пикселей
    double realScale((realMax-realMin)/(imageWidth-1));
    double imaginaryScale((imaginaryMax-imaginaryMin)/(imageHeight-1));

    // Лямбда-выражение для создания строки изображения
    std::function<void(HDC&, int)> rowCalc = [&] (HDC& memDC, int yLocal)
    {
        double zReal(0.0), cReal(0.0);
        double zImaginary(imaginaryMax - yLocal*imaginaryScale);
        double cImaginary(zImaginary);
        // Перебрать пиксели в строке
        for(int x = 0 ; x < imageWidth ; ++x)
        {
            zReal = cReal = realMin + x*realScale;

            // Установить цвет пикселя на основании n
            SetPixel(memDC, x, 0, Color(IteratePoint(zReal, zImaginary,
                                                     cReal, cImaginary)));
        }
    };
    // Перебрать строки изображения
    for(int y = 1 ; y < imageHeight ; y += 2)
    {
        Concurrency::parallel_invoke([&]{rowCalc(memDC1, y-1);},
                                     [&]{rowCalc(memDC2, y);});

        // Передать строку пикселей в контекст устройства клиентской
        // области
        BitBlt(hdc, 0, y-1, imageWidth, 1, memDC1, 0, 0, SRCCOPY);
        BitBlt(hdc, 0, y, imageWidth, 1, memDC2, 0, 0, SRCCOPY);
    }

    // Если это нечетная строка, позаботиться о последней
    if(imageHeight%2 == 1)
    {
        rowCalc(memDC1, imageWidth-1);
        BitBlt(hdc, 0, imageHeight-1, imageWidth, 1, memDC1,
               0, 0, SRCCOPY);
    }
    SelectObject(memDC1, oldBmp1);
    SelectObject(memDC2, oldBmp2);
    DeleteObject(bmp1);   // Удалить изображение 1
    DeleteObject(bmp2);   // Удалить изображение 2
    DeleteDC(memDC1);     // и рабочий DC 1
    DeleteDC(memDC2);     // и рабочий DC 2
    ReleaseDC(hWnd, hdc);  // Освободить клиентскую область DC
}
```

Новые и измененные фрагменты кода функции `DrawSet()` выделены полужирным шрифтом. Как можно заметить, здесь довольно много отличий от первоначальной версии функции. Теперь в памяти создаются два контекста устройства и два растровых изображения, по одному для каждой из двух параллельных задач. Если на вашем компьютере есть четыре ядра, можете создать по четыре каждого из них.

Поскольку код выполняющихся параллельно задач одинаков, чтобы не повторять его для каждого из аргументов алгоритма parallel_invoke, определите переменную функции rowCalc, инкапсулирующую его. Для этого используется шаблон function, описанный в главе 10 и определенный в заголовке functional. Аргументами объекта rowCalc являются контекст устройства в памяти и значение yLocal, являющееся индексом вычисляемой строки. Переменные, хранящие компоненты *Z* и *c*, теперь находятся в функции rowCalc, поскольку они не могут быть совместно использованы параллельными процессами. Функция rowCalc перебирает пиксели строки, как и прежде, и устанавливает цвет пикселя в переданном ему контексте устройства, расположенном в памяти, используя значение, которое возвратила функция IteratePoint().

В цикле по строкам, индексируемым переменной y, значение теперь начинается с 1 и продолжается с шагом 2, так как предстоит вычислять строки y и y-1 параллельно. Для этого используется алгоритм parallel_invoke. Аргументами алгоритма являются лямбда-выражение, вызывающее функцию rowCalc с аргументами, которые выбирают используемый контекст устройства, и значение y. По завершении алгоритма parallel_invoke вы последовательно передаете пиксели из каждого контекста устройства, расположенного в памяти, в контекст устройства окна hdc.

Если значение переменной imageHeight является нечетным, то последняя строка не будет обработана, поэтому ее придется отработать как частный случай, чтобы закончить изображение. Это подразумевает вызов функции rowCalc для последней строки со значением индекса imageHeight-1. И наконец, когда вычисление закончено, следует удалить контексты устройств, созданные в памяти, и растровые изображения.

Прежде чем компилировать и выполнить новую версию, необходимо добавить прототип функции DrawSetParallel() в файл исходного кода и изменить действие для сообщения WM_PAINT в функции WndProc(), чтобы вызвать эту новую функцию. Необходимо также добавить следующие директивы #include.

```
#include "ppl.h"
#include <functional> // Для шаблона function<>
```

При запуске новой версии программы должен получиться тот же образ набора Мандельброта, но уже быстрее. Было бы хорошо узнать, насколько быстрее, не так ли?

Хронометраж выполнения

Добавим в приложение класс, определяющий высокоточный таймер, который можно запустить в начале вычислений и остановить, когда они закончатся. Эта версия находится в загружаемом коде как пример Ex13_02C. Добавим также возможность использовать либо функцию DrawSet() для последовательного выполнения, либо функцию DrawSetParallel(), чтобы увидеть, как долго они выполняются. Переключайтесь между последовательным и параллельным выполнением, щелкая правой кнопкой мыши. Хорошее место для отображения времени — в строке состояния внизу окна приложения, поэтому добавим в приложение и ее.

Определение класса таймера

Класс HRTimer будет использовать функцию API Windows для обращения к высокоточному таймеру. Не у всех систем есть высокоточный таймер, поэтому если у вас его нет, пример не сработает. Я объясню основы работы этого класса, а исследование мелких деталей оставляю вам.

Добавьте в приложение новый файл заголовка, HRTimer.h и вставьте в него следующий код.

```
#pragma once
#include "windows.h"

class HRTimer
{
  public:
    // Конструктор
    HRTimer(void) : frequency(1.0 / GetFrequency()) { }

    // Получить частоту таймера (срабатываний в секунду)
    double GetFrequency(void)
    {
        LARGE_INTEGER proc_freq;
        ::QueryPerformanceFrequency(&proc_freq);
        return static_cast<double>(proc_freq.QuadPart);
    }

    // Запуск таймера
    void StartTimer(void)
    {
        // Установка потока для использования процессора 0
        DWORD_PTR oldmask = ::SetThreadAffinityMask(::GetCurrentThread(),
                                                    0);
        ::QueryPerformanceCounter(&start);
        // Восстановить исходный
        ::SetThreadAffinityMask(::GetCurrentThread(), oldmask);
    }

    // Остановить таймер и возвратить прошедшее время в секундах
    double StopTimer(void)
    {
        // Установка потока для использования процессора 0
        DWORD_PTR oldmask =
                          ::SetThreadAffinityMask(::GetCurrentThread(),0);
    ::QueryPerformanceCounter(&stop);
    // Восстановить исходный
    ::SetThreadAffinityMask(::GetCurrentThread(), oldmask);
    return ((stop.QuadPart - start.QuadPart) * frequency);
    }
  private:
    LARGE_INTEGER start; // Хранит время запуска
    LARGE_INTEGER stop;  // Хранит время остановки
    double frequency;    // Период одного сигнала таймера в секундах
};
```

У высокоточного таймера есть частота (количество срабатываний в секунду), которая может быть разной у разных процессоров. Поэтому перед использованием таймера следует узнать его частоту с помощью функции `::QueryPerformanceFrequency()`, которая сохраняет результат в переменной `LARGE_INTEGER`, переданной как аргумент. Вызовите эту функцию в функции-члене `GetFrequency()`, которая вызывается конструктором класса при инициализации переменной-члена `frequency`. Эта переменная хранит время в секундах между сигналами системного таймера как значение типа `double`; оно вычисляется как обратная величина частоты таймера. Тип `LARGE_INTEGER` — это 8-байтовое объединение, которое определено в файле заголовка `Windows.h`. Оно объединяет

как члены две структуры, подробности которых я не буду рассматривать, поскольку мы не будем использовать их, а также переменную-член QuadPart типа знакового 64-битового целого числа.

Создав объект HRTimer, запускаем таймер при вызове его функции StartTimer(). Функция ::QueryPerformanceCounter() возвращает начальное время и сохраняет его в переменной-члене start. Вызов функции ::SetThreadAffinityMask() перед получением начального времени гарантирует, что вызов осуществляется для конкретного процессора (процессора 0). Функция-член StopTimer() записывает время остановки и возвращает время выполнения на том же процессоре. Это должно устранить проблемы, связанные с вероятностью выполнения этих вызовов на разных процессорах.

Чтобы остановить таймер, вызовите его функцию StopTimer(). Эта функция вычисляет время, прошедшее от момента вызова функции StartTimer(), за счет разницы между двумя значениями (переменные QuadPart объектов stop и start типа 64-битовых целых чисел со знаком), и умноженной на значение переменной-члена frequency. В результате получится прошедшее время в секундах, возвращаемое как значение типа double.

Чтобы использовать объект HRTimer, добавьте в пример Ex13_02.cpp файла исходного кода директиву #include для файла заголовка HRTimer.h. Объект таймера следует создать в глобальной области видимости, добавив следующий оператор, в раздел глобальных определений файла.

```
HRTimer timer; // Объект таймера для вычисления времени процессора
```

Код применения таймера мы добавим после добавления строки состояния к окну приложения.

Добавление строки состояния

Строка состояния, как большинство объектов, отображаемых операционной системой Windows, является всего лишь еще одним окном, поэтому создаем строку состояния, вызвав функцию API CreateWindowEx(). Так как место кода ее создания находится в функции InitInstance(), добавьте следующий код непосредственно перед вызовом функции ShowWindow().

```
HWND hStatus = CreateWindowEx(0, STATUSCLASSNAME, NULL,
                   WS_CHILD | WS_VISIBLE | SBARS_SIZEGRIP, 0, 0, 0, 0,
                   hWnd, (HMENU)IDC_STATUS, GetModuleHandle(NULL), NULL);
if(!hStatus)      // Проверить, имеется ли строка состояния,
    return FALSE; // и возвратить FALSE, если нет

// Установить панели строки состояния
int statwidths[] = {100,200, -1};
SendMessage(hStatus, SB_SETPARTS, sizeof(statwidths)/sizeof(int),
                                  (LPARAM)statwidths);
SendMessage(hStatus, SB_SETTEXT, 0, (LPARAM)_T("Processor Time :"));
SendMessage(hStatus, SB_SETTEXT, 2,
                     (LPARAM)_T(" Right-click for parallel execution"));
```

Второй аргумент функции CreateWindowEx() определяет, что окно будет строкой состояния со стандартным именем класса STATUSCLASSNAME. Существует множество других имен стандартных классов для других видов элементов управления (они определены в файле заголовка commctrl.h), таких как панели инструментов, кнопки и т.д. Строка состояния будет дочерним окном по отношению к главному окну, hWnd. Оно

будет видимо, и у него будет захват для изменения размера в правом конце строки состояния. Размер и позиция строки состояния будут заданы по умолчанию, т.е. в основании родительского окна с высотой, соответствующей стандартному шрифту. Символ IDC_STATUS, находящийся в десятом аргументе, является индивидуальной целочисленной константой, которая идентифицирует элемент управления строки состояния. Необходимо добавить его к ресурсам проекта. Откройте представление ресурсов и щелкните правой кнопкой мыши на имени файла .rc. В появившемся контекстном меню выберите пункт Select Resource Symbols... (Выбор символов ресурса...). В открывшемся диалоговом окне Resource Symbols (Символы ресурса) щелкните на кнопке New... (Новый...), введите имя **IDC_STATUS** и щелкните на кнопке OK. Щелкните на кнопке Close (Закрыть), чтобы закрыть диалоговое окно. Значение для нового символа будет выбрано автоматически, чтобы оно отличалось от значений существующих символов.

Каждый элемент в массиве statwidths определяет размер (в логических единицах) раздела строки состояния, в котором будет отображаться текст. Здесь массив определяет три раздела, причем ширина последнего составляет –1. Это означает, что он занимает все место, оставшееся от первых двух разделов. Функция SendMessage() посылает сообщение в окно, определенное ее первым аргументом, в данном случае в окно строки состояния. Установите разделы строки состояния, передавая ей сообщение SB_PARTS с третьим аргументом, определяющим количество разделов, и с четвертым аргументом, определяющим размеры разделов. Следующие два вызова функции SendMessage() запишут текст в соответствующие разделы строки состояния, которые индексируются слева, начиная с нуля. Первое сообщение SB_SETTEXT устанавливает текст, определенный четвертым аргументом, в разделе, определенным третьим аргументом. Последнее посылаемое сообщение устанавливает текст в третьем разделе. Мы будем использовать вторую часть строки состояния для отображения прошедшего времени в секундах, когда оно будет вычислено.

Для компиляции этого кода необходимо добавить в файл исходного кода директиву #include для файла заголовка commctrl.h. Если теперь запустить программу, то внизу окна приложения должна появиться строка состояния.

Переключение между последовательным и параллельным выполнением

Сначала добавьте в файл Ex13_02C.cpp глобальную переменную со следующим определением.

```
bool parallel(false);    // Для параллельного выполнения - True
```

Это индикатор, используемый для выбора функции рисования в клиентской области, при получении сообщения WM_PAINT. Можете переключать значение этой переменной при обработке сообщения WM_RBUTTONDOWN, посылаемого щелчком правой кнопкой мыши. Добавьте следующий раздел case в оператор switch функции WndProc().

```
case WM_RBUTTONDOWN:
    parallel = !parallel; // Переключить индикатор
    // Обновить сообщение в строке состояния
    SendMessage(GetDlgItem(hWnd, IDC_STATUS), SB_SETTEXT, 2,
                    (LPARAM)(parallel ?
                    _T(" Right-click for serial execution") :
                    _T(" Right-click for parallel execution")));
    InvalidateRect (hWnd, NULL, TRUE);
    UpdateWindow (hWnd);
    break;
```

Здесь вы переключаете состояние переменной parallel и посылаете строке состояния сообщение о модификации текста в третьем разделе строки состояния, отображающем текущее значение переменной parallel. Вызов функции InvalidateRect() определяет область окна, заданного первым аргументом, который должен быть перерисован, когда будет получено следующее сообщение WM_PAINT. Второй аргумент — это объект RECT, определяющий регион клиентской области, который должен быть перерисован, но значение NULL определяет здесь всю клиентскую область окна. Третий аргумент — TRUE, если клиентская область должна быть стерта прежде, чем быть перерисованной. Вызов функции UpdateWindow() посылает сообщение WM_PAINT окну приложения, которое приведет к повторному вычислению набора Мандельброта последовательным или параллельным способом.

Теперь можно изменить код, который обрабатывает сообщение WM_PAINT, чтобы нарисовать клиентскую область окна.

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
// TODO: Здесь добавить весь код рисования...
if(parallel)
    DrawSetParallel(hWnd);
else
    DrawSet(hWnd);
EndPaint(hWnd, &ps);
break;
```

Обработка сообщения подразумевает теперь выбор функции, используемой для вычисления набора Мандельброта, на основании значения переменной parallel. Когда значением переменной parallel является true, используется параллельная версия функции, а когда false — последовательная.

Использование таймера

Мы можем запустить и остановить глобальный таймер, созданный в коде, обрабатывающем сообщение WM_PAINT, но есть небольшая проблема. Как вы видели ранее, объект HRTimer возвращает прошедшее время как значение типа double. Его отображение в строке состояния требует, чтобы это значение было в виде текстовой строки с нулевым символом в конце. Первый шаг в использовании таймера подразумевает добавление функции, которая получает время как значение типа double и обновляет строку состояния с текстовым представлением значения типа double.

Добавьте следующий прототип в файл Ex13_02C.cpp после уже существующих прототипов функций.

```
void UpdateStatusBarTime(HWND hWnd, double time);
```

Ниже приведено определение функции, которую следует добавить в файл исходного кода.

```
void UpdateStatusBarTime(HWND hWnd, double time)
{
    std:: basic_ stringstream <TCHAR> ss; // Создать строковый поток
    // Включить fixed и выключить float
    ss.setf(std::ios::fixed, std::ios::floatfield);
    ss.precision(2);                // Две позиции после десятичной точки
    ss << _T(" ") << time << _T(" seconds"); // Запись времени в ss
    SendMessage(GetDlgItem(hWnd, IDC_STATUS), SB_SETTEXT, 1,
                (LPARAM)(ss.str().c_str()));
}
```

Основную работу выполняет объект класса basic_stringstream, шаблон которого определен в заголовке sstream; таким образом, необходимо добавить для него директиву #include. Объект basic_stringstream<TCHAR> является потоком, передающим записанные в него данные, в строковый буфер, хранящий символы типа TCHAR. Это весьма полезно для операций форматирования. Здесь для объекта ss устанавливаются флаги передачи значений с плавающей запятой основному буферу в представлении с фиксированной запятой вместо стандартного научного представления. Первый аргумент функции setf() — это устанавливаемые флаги, второй аргумент — сбрасываемые флаги. Вызов функции precision() для объекта ss с аргументом 2 задает для значений с плавающей запятой две позиции после десятичной запятой. Затем значение времени записывается в объект ss наряду с описательным текстом.

Обновим строку состояния значением времени, вызвав функцию SendMessage(). Последний аргумент — вызов функции str() для объекта ss, чтобы получить содержимое потока как объект string. Для этого вызов функции c_str() создает строку с завершающим нулевым символом. Аргумент строкового указателя должен иметь здесь тип LPARAM, таким образом, вы приводите указатель для этой строки к типу LPARAM.

Последний этап — коррекция обработчика сообщения WM_PAINT в функции WndProc(), чтобы использовался таймер.

```
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Здесь добавить весь код рисования...
    timer.StartTimer();
    if(parallel)
        DrawSetParallel(hWnd);
    else
        DrawSet(hWnd);
    UpdateStatusBarTime(hWnd, timer.StopTimer());
    EndPaint(hWnd, &ps);
    break;
```

Теперь нужно отобразить время в строке состояния. Щелкая правой кнопкой мыши, можно переключаться между последовательным и параллельным выполнением. Это позволит выяснить, насколько увеличится производительность при переходе на параллельное выполнение. На рис 13.3 показано окно приложения при параллельном выполнении на моей машине.

Результат получен на три с лишним секунды быстрее, чем при последовательном вычислении на моем компьютере. Имейте в виду, что другие программы, выполняющиеся на вашем компьютере, могут существенно повлиять на время выполнения. Кроме того, различные системные службы операционных систем и антивирусное программное обеспечение, выполняющиеся в фоновом режиме, также окажут влияние, поэтому не ожидайте, что время выполнения, измеренное в данном примере, будет обязательно одинаковым в разных случаях. Вы найдете различные рабочие этапы разработки этого примера в загружаемом коде для книги под именами Ex13_02A, Ex13_02B и Ex13_02C.

А как же алгоритм parallel_for, описанный не так давно в настоящей главе? Можем ли мы применить его для ускорения вычисления набора Мандельброта, с учетом того, что вычисление почти полностью состоит из циклов for? Конечно, мы можем сделать внутренний цикл параллельным при создании нового объекта memDC для каждой линии изображения, но было бы неплохо сделать параллельным внешний цикл. Но вызов функции BitBlt(), который должен находиться во внешнем цикле, не может быть выполнен параллельно, поскольку он осуществляет запись в контекст устройства.

Это действительно препятствует использованию алгоритма `parallel_for` для внешнего цикла. Тем не менее с небольшой помощью библиотеки PPL мы, возможно, сможем устранить проблему.

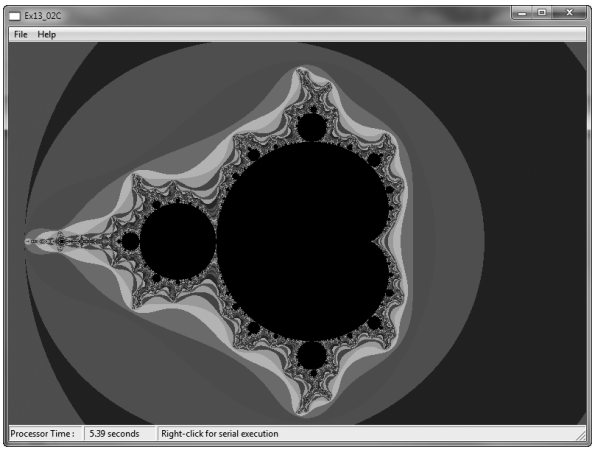

Рис. 13.3. Окно приложения при параллельном выполнении

Критические разделы

Критический раздел — это непрерывная последовательность операторов, которая не может выполняться несколькими задачами одновременно. Как вы видели в последнем примере, параллельное выполнение кода, модифицирующего контекст устройства, вызывает проблемы, поскольку контекст устройства не предназначен для использования таким образом. Критические разделы могут находиться и в других контекстах.

Библиотека PPL определяет класс `critical_section`, который можно использовать для идентификации последовательностей операторов, которые не должны выполняться параллельно.

Использование объекта `critical_section`

У класса `critical_section` есть только один конструктор, поэтому создать его объект очень просто.

```
Concurrency::critical_section cs;
```

Это создает объект `cs`. Объект класса `critical_section` — это *мьютекс* (mutex — сокращение от *mut*ual *ex*clusion (взаимное исключение)). Мьютекс позволяет нескольким потокам выполнения получать доступ к конкретному ресурсу, но только по одному. Это механизм, позволяющий потоку блокировать ресурс, который он в настоящее время использует. Все другие потоки должны подождать, пока ресурс не будет разблокирован, чтобы обратиться к нему.

Блокировка и разблокировка разделов кода

Как только объект `critical_section` будет создан, можете применять блокировку к разделу кода, вызвав его функцию-член `lock()` следующим образом.

```
cs.lock();   // Блокировать следующие операторы
```

Операторы, которые следуют за вызовом функции-члена `lock()`, могут быть выполнены только одним потоком одновременно. Все другие потоки исключаются до вызова функции-члена `unlock()` мьютекса.

```
cs.lock();   // Блокировать следующие операторы

// Код для одного потока...

cs.unlock(); // Освободить блокировку
```

Как только вызван метод `unlock()` объекта класса `critical_section`, другой поток может применить блокировку к разделу кода и выполнить его.

Если задача вызывала метод `lock()` объекта класса `critical_section` и код уже выполняется другой задачей, следующий вызов метода `lock()` блокируется. Другими словами, выполнение задачи, пытающейся блокировать раздел кода, не может продолжаться. Это нередко оказывается неудобным. Например, если несколько разных разделов кода контролируется разными мьютексами, то вы могли бы выполнить некий раздел кода, если другой раздел не блокируется. Если же вы не хотите блокировать раздел кода и есть возможность выполнять другой код, используйте функцию-член `try_lock()`.

```
if(cs.try_lock())
{
    // Код для одного потока...
    cs.unlock(); // Освободить блокировку
}
else
    // Делать что-то еще...
```

Функция `try_lock()` не осуществляет блокировку, если она невозможна, но возвращает логическое значение `true`, если блокировка получена, и значение `false` в противном случае.

Полагаю, теперь имеется достаточно возможностей, чтобы преодолеть проблемы применения алгоритма `parallel_for` при вычислении набора Мандельброта.

Практическое занятие

Использование алгоритма `parallel_for` для набора Мандельброта

Можете просто исправить предыдущий пример, заменив функцию `DrawSetParallel()` новой версией. В загружаемом коде это пример `Ex13_03`.

Самый простой способ применения алгоритма `parallel_for` подразумевает замену внешнего цикла в первоначальной функции `DrawSet()` и последующую блокировку тех разделов кода, которые не должны выполниться одновременно. Придется также перестроить часть кода, чтобы обеспечить параллельные операции цикла. Код приведен ниже.

Доступно для загрузки на Wrox.com

```
void DrawSetParallel(HWND hWnd)
{
    HDC hdc(GetDC(hWnd)); // Получить контекст устройства

    // Получить размерности клиентской области, составляющие размер
    // изображения
    RECT rect;
```

```
    GetClientRect(hWnd, &rect);
    int imageHeight(rect.bottom);
    int imageWidth(rect.right);

    // Оси клиентской области
    const double realMin(-2.1); // Минимальное вещественное значение
    double imaginaryMin(-1.3);  // Минимальное мнимое значение
    double imaginaryMax(+1.3);  // Максимальное мнимое значение

    // Установить максимум вещественного значения, чтобы оси имели
    // одинаковый масштаб
    double realMax(realMin+(imaginaryMax-
                            imaginaryMin)*imageWidth/imageHeight);

    // Получить коэффициенты масштабирования для координат пикселей
    double realScale((realMax-realMin)/(imageWidth-1));
    double imaginaryScale((imaginaryMax-imaginaryMin)/(imageHeight-1));
    Concurrency::critical_section cs; // Мьютекс для операции BitBlt()
    // Параллельный перебор строк изображения
    Concurrency::parallel_for(0, imageHeight,[&](int y)
    {
        cs.lock(); // Блокировка доступа к клиентскому DC

        // Создать один ряд пикселей изображения
        HDC memDC = CreateCompatibleDC(hdc); // Получить контекст
                                // устройства для рисования пикселей
        HBITMAP bmp = CreateCompatibleBitmap(hdc, imageWidth, 1);
        cs.unlock();            // С hdc закончено, разблокировать
        HGDIOBJ oldBmp = SelectObject(memDC, bmp); // Выбрать изображение
                                                   // в DC
        double cReal(0.0), cImaginary(0.0); // Сохранить компоненты c
        double zReal(0.0), zImaginary(0.0); // Сохранить компоненты z
        zImaginary = cImaginary = imaginaryMax - y*imaginaryScale;
        // Перебрать пиксели в строке
        for(int x = 0 ; x < imageWidth ; ++x)
        {
            zReal = cReal = realMin + x*realScale;

            // Установить цвет пикселя на основании n
            SetPixel(memDC, x, 0, Color(IteratePoint(zReal, zImaginary,
                                                     cReal, cImaginary)));
        }
        cs.lock(); // Блокировка для записи в hdc

        // Передать строку пикселей в контекст устройства клиентской
        // области
        BitBlt(hdc, 0, y, imageWidth, 1, memDC, 0, 0, SRCCOPY);
        cs.unlock(); // Освободить блокировку
        SelectObject(memDC, oldBmp);
        DeleteObject(bmp);      // Удалить изображение
        DeleteDC(memDC);        // и рабочий DC
    });
    ReleaseDC(hWnd, hdc);       // Освободить клиентскую область DC
}
```

Фрагмент кода Ex13_03.cpp

Когда вы заменяете этой функцией одноименную функцию в предыдущем примере, все должно быть готово. Запуск версии, использующей алгоритм `parallel_for`, демонстрирует непоследовательный характер операций с различными строками изображения, отображаемыми в разное время. На моем компьютере этот код, похоже, работал немного быстрее, чем версия, использующая алгоритм `parallel_invoke`.

Шаблон класса `combinable`

Шаблон класса `combinable` предназначен для помощи в совместном использовании ресурсов, которые могут быть сегментированы в локальные переменные для потоков и объединены после завершения вычисления. Это предоставляет преимущества перед использованием мьютекса как объекта `critical_section`, в котором вы не должны блокировать совместно используемые ресурсы и вынуждать потоки ждать. Объект `combinable` способен создавать дубликаты совместно используемых ресурсов, по одному для каждого потока, который должен быть выполнен параллельно. Каждый поток использует собственную независимую копию исходного совместно используемого ресурса. Единственное условие — вам следует определить операцию, которая объединит дубликаты, чтобы сформировать единый ресурс, когда параллельная операция закончится. Очень простым примером, где может помочь класс `combinable`, является переменная, которая совместно используется в цикле для суммирования общего значения. Рассмотрим следующий фрагмент кода.

```
std::array<double, 100> values;
double sum(0.0);
for(size_t i = 0 ; i<values.size() ; ++i)
{
    values[i] = (i+1)*(i+1);
    sum += values[i];
}
std::cout << "Total = " << sum << std::endl;
```

Вы не сможете сделать этот цикл параллельным, не сделав чего-нибудь с переменной `sum`, поскольку к ней обращаются и модифицируют при каждой итерации. Разрешение параллельных потоков, вероятно, повредит значение результата. Ниже показано, как класс `combinable` поможет вам выполнить цикл параллельно.

```
Concurrency::combinable<double> sums;
Concurrency::parallel_for(static_cast<size_t>(0), values.size(),
                          [&values, &sums](size_t i)
                          {
                              values[i] = (i+1)*(i+1);
                              sums.local() += values[i];
                          });
double sum = sums.combine([](double lSum, double rSum)
                          { return lSum + rSum; });
```

Объект `sums` класса `combinable` может создать локальную переменную типа `double` для каждого параллельного потока. Аргумент шаблона определяет тип переменной. Каждый параллельный поток использует собственную локальную переменную определенного типа, к которой вы обращаетесь в лямбда-выражении при вызове функции `local()` для объекта `sums`.

Как только выполнится цикл `parallel_for`, объедините локальные переменные, принадлежащие объекту `sums`, вызвав его функцию `combine()`. Аргумент является

лямбда-выражением, которое определяет, как должны быть объединены две любые локальные переменные. У лямбда-выражения должно быть два параметра, которые имеют одинаковый тип, `T`, использованный для аргумента шаблона `combinable`, или константные ссылки на тип `T`. Лямбда-выражение должно возвращать результат объединения двух своих аргументов. Функция `combine()` будет использовать лямбда-выражение по мере необходимости, чтобы объединить все локальные переменные, которые были созданы, и возвратит результат. В данном случае это будет сумма всех элементов `values`.

Класс `combinable` определяет также функцию-член `combine_each()`, чтобы объединять локальные результаты, типом возвращаемого значения которых является `void`. Аргумент этой функции должен быть лямбда-выражением или объектом функции с одним параметром типа `T` или `const T&`. Эта функция позволяет объединить результат. Ниже показано, как ее можно использовать для объединения локальных результатов предыдущего фрагмента кода.

```
double sum(0.0);
sums.combine_each([&sum](double localSum)
                  { sum += localSum; });
```

Лямбда-выражение, являющееся аргументом функции `combine_each()`, будет вызвано по одному разу для каждой локальной части общей суммы, принадлежащей объекту `sums`. Таким образом, общий итог будет просуммирован в переменной `sum`.

Объединяющийся объект можно использовать многократно. Чтобы сбросить значение локальных переменных для объединяющегося объекта, достаточно вызвать его функцию `clear()`.

Практическое занятие

Использование класса `combinable`

Рассмотрим использование класса `combinable` на примере суммирования известной прогрессии, приближающейся к значению π^2.


Доступно для загрузки на Wrox.com

```
// Ex13_04.cpp
// Вычисление пи в результате суммирования прогрессии
#include <iostream>
#include <iomanip>
#include <cmath>
#include "ppl.h"

int main()
{
    Concurrency::combinable<double> piParts;
    Concurrency::parallel_for(1, 1000000, [&piParts](long long n)
                              { piParts.local() += 6.0/(n*n); });

    double pi2 = piParts.combine([](double left, double right)
                                 { return left + right; });
    std::cout << "pi squared = " << std::setprecision(10) << pi2
              << std::endl;
    std::cout << "pi = " << std::sqrt(pi2) << std::endl;
    return 0;
}
```

Результат получается таким.

```
pi squared = 9.869598401
pi = 3.141591699
```

Как это работает

Сначала мы создаем объект `piParts` класса `combinable` для использования в алгоритме. Алгоритм `parallel_for` суммирует один миллион элементов прогрессии.

```
,6-,1-2..+,6-,2-2..+,6-,3-2..+ ...
```

Алгоритм выполнит итерации параллельно на таком количестве процессоров, которое есть на машине. Сумма элементов, вычисленных на каждом процессоре, накапливается в локальной копии переменной типа `double`, предоставляемой объектом `piParts` класса `combinable`. После завершения работы алгоритма мы суммируем локальные переменные, вызвав функцию `combine()` объекта `piParts`. Аргумент — лямбда-выражение, определяющее способ объединения двух локальных значений. Функция `combine()` возвращает сумму всех используемых локальных переменных. Чтобы получить искомое значение, необходимо извлечь квадратный корень из результата.

Задачи и группы задач

Вы можете использовать объект класса `task_group` для параллельного решения двух и более задач, где задача может быть определена функтором или лямбда-выражением. Объект `task_group` потокобезопасен, таким образом, можете использовать его для инициализации задачи для разных потоков. Если захотите инициализировать параллельные задачи от одного потока, то объект `structured_task_group` будет более эффективным.

Чтобы выполнить задачу в другом потоке, сначала создайте собственный объект `task_group`; затем передайте объект функции или лямбда-выражению, которые определяют задачу для функции `run()` объекта. Фрагмент кода, демонстрирующий, как это работает, приведен ниже.

```
std::array<int, 50> odds;
std::array<int, 50> evens;
int n(0);
std::generate(odds.begin(), odds.end(), [&n] { return ++n*2 - 1; });
n = 0;
std::generate(evens.begin(), evens.end(), [&n] { return ++n*2; });
int oddSum(0), evenSum(0);
Concurrency::task_group taskGroup;
taskGroup.run([&odds, &oddSum]
              { for(size_t i = 0 ; i<odds.size() ; ++i)
                    oddSum += odds[i];
              });
for(size_t i = 0 ; i< evens.size() ; ++i)
    evenSum += evens[i];
taskGroup.wait();
```

Массивы `odds` и `evens` (четные и нечетные) инициализируются соответственно с последовательными четными и нечетными целыми числами. Переменные `oddSum` и `evenSum` используются для накапливания суммы элементов двух массивов. Первый вызов функции `run()` для объекта `taskGroup` инициализирует задачу, определенную лямбда-выражением, в потоке, отдельном от текущего потока, — другими словами, на отдельном процессоре. Лямбда-выражение суммирует четные элементы и накапливает результат в переменной `oddSum`. Цикл `for` следует за суммированием элементов массива `evens` и выполняется одновременно с группой задач лямбда-выражения. Вызов

функции wait() для объекта taskGroup приостанавливает выполнение текущего потока, пока все задачи, инициализированные объектом taskGroup (в данном случае только один), не будут завершены.

Текущий поток выполнения может возобновиться, как только будет инициализирована задача группы задач. Когда вы вызовете функцию wait() для объекта task_group, текущий поток только приостановит выполнение. Конечно, если у вас есть больше двух процессоров, можете использовать объект task_group для инициализации более чем одной задачи, выполняющейся одновременно с текущим потоком.

Объект structured_task_group может использоваться только для инициализации задачи одного потока. В отличие от объектов task_group вы не сможете передать лямбда-выражение или объект функции в функцию run() для объекта structured_task_group. Аргументом, определяющим задачу, в данном случае будет объект task_handle. Это шаблон класса, где параметром типа является объект функции, который он инкапсулирует. Создать объект шаблона task_handle довольно просто. Ниже показано, как предыдущий фрагмент кода может быть переписан для использования объекта structured_task_group, инициализирующего параллельную задачу.

```
std::array<int, 50> odds;
std::array<int, 50> evens;
int n(0);
std::generate(odds.begin(), odds.end(), [&n] { return ++n*2 - 1; });
n = 0;
std::generate(evens.begin(), evens.end(), [&n] { return ++n*2; });
int oddSum(0), evenSum(0);
Concurrency::structured_task_group taskGroup;
auto oddsFun = [&odds, &oddSum]
{ for(size_t i = 0 ; i<odds.size() ; ++i)
      oddSum += odds[i];
};
taskGroup.run(Concurrency::task_handle<decltype(oddsFun)>(oddsFun));
for(size_t i = 0 ; i< evens.size() ; ++i)
    evenSum += evens [i];
taskGroup.wait();
```

Здесь создается объект функции, oddsFun, из лямбда-выражения. Ключевое слово auto получает правильный тип для объекта функции oddsFun из лямбда-выражения. Функция run() для объекта taskGroup требует, чтобы аргумент был объектом task_handle, инкапсулирующим инициализируемую задачу. Вы создаете его при передаче объекта oddsFun как аргумента, с аргументом типа для шаблона task_handle, определенного как объявленный тип для объекта oddsFun. Для этого используется ключевое слово decltype.

Обратите внимание на то, что аргументом конструктора объекта task_handle должен быть объект функции, у которого нет никаких параметров. Следовательно, вы не сможете использовать здесь лямбда-выражение, которому при выполнении требуются аргументы. Если вы действительно должны использовать лямбда-выражение с параметрами, заключите его в другое лямбда-выражение без параметров, которое вызывает исходное лямбда-выражение.

Еще один важный момент: вы должны гарантировать, что объект task_handle не будет ликвидирован прежде, чем связанная с ним задача закончит выполнение. Это может случиться, если выполнение текущего потока, в котором был создан объект task_handle и инициализировано параллельное выполнение задачи, продолжится до места вызова деструктора объекта task_handle, в то время как параллельная задача все еще выполняется.

Практическое занятие

Использование группы задач для вычисления набора Мандельброта

Создадим новую версию прежнего примера вычисления набора Мандельброта, которая будет использовать группу задач. Я представлю здесь только код функции `DrawSetParallel()`, поскольку это единственный отличающийся фрагмент. Остальная часть кода будет точно такой же; полный код примера находится в загружаемом коде под именем `Ex13_05`. Вот как может быть реализована функция с использованием группы задач.

```
void DrawSetParallel(HWND hWnd)
{
    HDC hdc(GetDC(hWnd)); // Получить контекст устройства

    // Получить размерности клиентской области, составляющие размер
    // изображения
    RECT rect;
    GetClientRect(hWnd, &rect);
    int imageHeight(rect.bottom);
    int imageWidth(rect.right);

    // Создать один ряд пикселей изображения
    HDC memDC1 = CreateCompatibleDC(hdc); // Получить контекст устройства
                                          // для рисования пикселей
    HDC memDC2 = CreateCompatibleDC(hdc); // Получить контекст устройства
                                          // для рисования пикселей
    HBITMAP bmp1 = CreateCompatibleBitmap(hdc, imageWidth, 1);
    HBITMAP bmp2 = CreateCompatibleBitmap(hdc, imageWidth, 1);
    HGDIOBJ oldBmp1 = SelectObject(memDC1, bmp1); // Выбрать изображение
                                                  // в DC1
    HGDIOBJ oldBmp2 = SelectObject(memDC2, bmp2); // Выбрать изображение
                                                  // в DC2
    // Оси клиентской области
    const double realMin(-2.1); // Минимальное вещественное значение
    double imaginaryMin(-1.3);  // Минимальное мнимое значение
    double imaginaryMax(+1.3);  // Максимальное мнимое значение

    // Установить максимум вещественного значения, чтобы оси имели
    // одинаковый масштаб
    double realMax(realMin+(imaginaryMax-
                            imaginaryMin)*imageWidth/imageHeight);

    // Получить коэффициенты масштабирования для координат пикселей
    double realScale((realMax-realMin)/(imageWidth-1));
    double imaginaryScale((imaginaryMax-imaginaryMin)/(imageHeight-1));

    // Лямбда-выражение для создания строки изображения
    auto rowCalc = [&] (HDC& memDC, int yLocal)
    {
        double zReal(0.0), cReal(0.0);
        double zImaginary(imaginaryMax - yLocal*imaginaryScale);
        double cImaginary(zImaginary);
        // Перебрать пиксели в строке
```

```
        for(int x = 0 ; x < imageWidth ; ++x)
        {
            zReal = cReal = realMin + x*realScale;

            // Установить цвет пикселя на основании n
            SetPixel(memDC, x, 0, Color(IteratePoint(zReal, zImaginary,
                                                     cReal, cImaginary)));
        }
    };

    Concurrency::task_group taskGroup;
    // Перебрать строки изображения
    for(int y = 1 ; y < imageHeight ; y += 2)
    {
        taskGroup.run([&]{rowCalc(memDC1, y-1);});
        rowCalc(memDC2, y);
        taskGroup.wait();
        BitBlt(hdc, 0, y-1, imageWidth, 1, memDC1, 0, 0, SRCCOPY);
        BitBlt(hdc, 0, y, imageWidth, 1, memDC2, 0, 0, SRCCOPY);
    }

    // Если это нечетная строка, позаботиться о последней
    if(imageHeight%2 == 1)
    {
        rowCalc(memDC1, imageWidth-1);
        BitBlt(hdc, 0, imageHeight-1, imageWidth, 1, memDC1, 0, 0,
               SRCCOPY);
    }
    SelectObject(memDC1, oldBmp1);
    SelectObject(memDC2, oldBmp2);
    DeleteObject(bmp1);   // Удалить изображение 1
    DeleteObject(bmp2);   // Удалить изображение 2
    DeleteDC(memDC1);     // и рабочий DC 1
    DeleteDC(memDC2);     // и рабочий DC 2
    ReleaseDC(hWnd, hdc); // Освободить клиентскую область DC
}
```

Вывод будет таким же, как у прежней версии программы.

Как это работает

Изменения в коде функции выделены полужирным шрифтом; отличаются только четыре строки. После создания объекта `task_group` вне цикла мы передаем лямбда-выражение функции `run()`, чтобы выполнить его в другом потоке. Лямбда-выражение, являющееся аргументом, вызывает объект функции `rowCalc`, который определяется другим лямбда-выражением. Объект функции `rowCalc` вычисляет пиксели для одной строки в клиентской области окна, используя значение и расположенный в памяти контекст устройства памяти, которые передаются как аргументы. Это нужно, чтобы поместить объект `rowCalc` в другое лямбда-выражение, поскольку аргументом функции `run()` должен быть объект функции, который не требует аргументов при выполнении.

После вызова функции `run()` для объекта `taskGroup` вы запускаете выполнение объекта функции `rowCalc` для следующей строки пикселей изображения. Это вычисление выполняется одновременно с вычислением, инициализированным функцией `run()` для объекта `taskGroup`. Вызов функции `wait()` для объекта `taskGroup`

приостанавливает выполнение текущего потока, пока не завершится задача, выполняемая объектом taskGroup. Теперь можно передать пиксели из контекстов устройств, расположенных в памяти, в контекст устройства клиентской области. Таким образом, цикл, индексированный переменной y, вычисляет две строки пикселей изображения, одновременно на каждой итерации.

Резюме

В этой главе вы изучили фундаментальные элементы, предоставляемые библиотекой шаблонов для параллельных вычислений, используемые для программ, которые применяют несколько процессоров. Библиотека PPL эффективна лишь при наличии существенных объемов вычислений, выполняемых на машине с несколькими процессорами, однако такие задачи отнюдь не редкость. Большинство инженерных и научных приложений входят в эту категорию наряду с множеством операций обработки изображения, и вы вполне можете найти крупномасштабные коммерческие задачи обработки данных, которые могут извлечь пользу из использования библиотеки PPL.

Упражнения

Исходные коды упражнений и их решения можно загрузить с веб-сайта www.wrox.com.

1. Определите функцию, которая вычислит факториал целого числа типа long long с использованием алгоритма parallel_for. (Факториалом целого числа *n* является произведение всех целых чисел от 1 до *n*.) Продемонстрируйте, что функция работает с соответствующей функцией main().
2. Определите функцию, которая параллельно вычислит сумму квадратов элементов контейнера array<double>, используя алгоритм parallel_invoke. Массив array<double> должен быть передан как аргумент функции. Продемонстрируйте, что функция работает с соответствующей функцией main(), сначала с ограниченным количеством известных значений, а затем с большим количеством произвольных значений.
3. Повторите предыдущее упражнение, но на сей раз используйте для параллельных вычислений объект task_group.

Что вы узнали в этой главе

- *Алгоритм* parallel_for. Предоставляет эквивалент цикла for с итерациями, выполняющимися параллельно. Задача, которая должна быть выполнена на каждой итерации, определяется с помощью объекта функции или лямбда-выражения.
- *Алгоритм* parallel_for_each. Предоставляет параллельный цикл, обрабатывающий содержимое контейнера STL с использованием итераторов.
- *Алгоритм* parallel_invoke. Используется для параллельного выполнения двух–десяти задач. Задача определяется с помощью объекта функции или лямбда-выражения.
- *Класс* combinable. Объект этого класса можно использовать для обслуживания ресурсов, к которым одновременно обращается несколько задач.
- *Класс* task_group. Объект этого класса можно использовать для выполнения одной или нескольких задач в отдельном потоке. Объект task_group потокобезопасен и применяется для инициализации задачи в другом потоке.

- *Класс* `structured_task_group`. Объект этого класса можно использовать для выполнения одной или нескольких задач в отдельных потоках. Объект `structured_task_group` не потокобезопасен, и все задачи следует инициализировать в том же потоке. Каждая задача должна быть идентифицирована с использованием объекта `task_handle`.
- *Высокоточный таймер.* Для реализации в приложении высокоточного таймера используйте функции `::QueryPerformanceFrequency()` и `::QueryPerformanceCounter()`, предоставляемые интерфейсом API Windows.