# Применение средств распараллеливания вычислений в реализации алгоритма вычисления фрактальной размерности двумерных изображений

И.Е. Красикова, В.В. Картузов, И.В. Красиков

***Аннотация*— В работе рассмотрено применение средств распараллеливания вычислений на языке программирования C++ (Windows API, OpenMP, C++11) к компьютерной реализации алгоритма вычисления фрактальной размерности двумерных изображений. Рассмотрены преимущества и недостатки указанных методов, показано, что наиболее простым и эффективным является применение стандартных средств C++11.**

***Ключевые слова*— алгоритм, фрактальная размерность, параллельные вычисления, OpenMP, C++11, динамическая многопоточность, многопроцессорные системы**

## I. Введение

В современном материаловедении фрактальные характеристики достаточно прочно заняли свое место при описании структурных свойств материалов. Однако несмотря на это, практически отсутствует какая-либо унификация методов определения их фрактальных и мультифрактальных характеристик. Это затрудняет работу исследователей в данной области, поскольку на практике оказывается достаточно сложно, если не вовсе невозможно, достоверно повторить опубликованные исследования.

Долгое время одной из наиболее общепризнанных программ такого рода занимала программа MFRDrom [1,2]. Однако, как показали детальные исследования этой программы на модельных объектах [3], данная программа обладает низкой устойчивостью, так что минимальные изменения исходного изображения могут привести к большим изменениям получаемых с ее помощью фрактальных характеристик.

Поэтому была проведена работа по программной реализации алгоритмов вычисления фрактальных характеристик двумерных изображений [4,5], в результате которой была разработана программа для получения мультифрактальных характеристик двумерных изображений. Проведенные исследования показали высокую устойчивость использованных алгоритмов и их реализации [6].

Несмотря на эффективную работу программы, с ростом размеров изображений время расчетов существенно увеличивается, так что (что характерно для любых задач с большим количеством вычислений) встает вопрос о повышении эффективности разработанного программного обеспечения.

Как правило, основным источником повышения быстродействия является применение другого, более эффективного алгоритма. Однако в настоящее время используется наиболее быстрый алгоритм Колмогорова, более производительной замены которому пока не найдено. Следующий шаг — оптимизация исходного кода программы, как программистом, так и оптимизирующим компилятором. И, наконец, повышение эффективности, связанное с применением более производительной вычислительной техники. Однако здесь также имеются свои проблемы.

В последнее время в связи с тем, что закон Мура [7] об экспоненциальном росте производительности вычислительной техники перестал адекватно описывать развитие вычислительной техники, стало необходимым применение параллелизма, в особенности для вычислительных задач. До сих пор, на протяжении многих лет, производители процессоров постоянно увеличивали тактовую частоту и параллелизм на уровне инструкций, так что на новых процессорах старые однопоточные приложения исполнялись быстрее без каких-либо изменений в программном коде. Сейчас по разным причинам производители процессоров предпочитают многоядерные архитектуры, и для получения всей выгоды от возросшей производительности центральных процессоров программы должны переписываться в соответствующей манере, с использованием параллельных алгоритмов. Для решения задач, в особенности хорошо подходящих для параллельных вычислений, когда задача может быть разделена на множество не взаимодействующих между собой процессов (например, обработка большого массива экспериментальных данных), так что временем обмена информацией между процессами по сравнению со временем вычислений можно пренебречь, можно использовать распределенные вычислительные системы

Красикова Ирина Евгеньевна, м.н.с. Института проблем материаловедения НАН Украины, e-mail: ira@ipms.kiev.ua

Картузов Валерий Васильевич, зав. отд. Института проблем материаловедения НАН Украины, e-mail: vvk@ipms.kiev.ua

Красиков Игорь Владимирович, с.н.с. Института проблем материаловедения НАН Украины, e-mail: kiv@kiv.kiev.ua

(грид).

Основная сложность при проектировании параллельных программ — обеспечение корректной последовательности взаимодействий между различными вычислительными процессами, а также координации ресурсов, разделяемых между процессами. В связи с этим проще всего распараллеливаются задачи, в которых вычисления можно разложить на не связанные между собой задачи, которые могут обращаться к одним и тем же данным для чтения, но не для записи. Классическим примером такой задачи является, например, перемножение матриц, когда каждый элемент результирующей матрицы может быть вычислен совершенно независимо от прочих элементов.

В той или иной форме многопроцессорные компьютеры существуют уже десятилетия. Но несмотря на это, до сих пор ни одна модель для параллельных вычислений не получила полного признания со стороны сообщества программистов. Тем не менее, можно считать, что одним из наиболее распространенных классов параллельных платформ является динамическая многопоточность [8].

В данной статье нами рассматривается возможность применения параллельных вычислений при определении мультифрактальных характеристик двумерных изображений и сравниваются три подхода к разработке многопоточного приложения на основе ранее описывавшейся компьютерной реализации алгоритма вычисления фрактальной размерности двумерных изображений [4].

## II. Исследование возможностей распараллеливания вычислений

В связи с тем, что развитие ранее использовавшегося в этой работе компилятора Open Watcom C++ практически остановлено и за последние пять лет не появилось никакой новой версии (при том, что за то же время язык программирования C++ был существенно обновлен стандартами C++11 и C++14), было принято решение о переходе на компилятор Visual C++ (обладающий к тому же более высокой степенью оптимизации); в настоящий момент используется версия Visual C++ 2013. В этом компиляторе многопоточность поддерживается на разных уровнях — как на наиболее низком, с помощью средств операционной системы (применением функций Windows API), так и на более высоком (с использованием открытого стандарта для распараллеливания программ OpenMP [9] и стандартных средств распараллеливания C++, появившихся в стандарте C++11 [10]).

Основное время при работе рассматриваемой в статье программы занимают декодирование исходного изображения (преобразование из конкретного графического формата наподобие JPG, TIFF и т.п. во внутреннее представление; преобразование изображения в черно-белое; построение гистограммы), а также просчет статистики с использованием пробных боксов. В связи с тем, что первая часть обработки выполняется с помощью вызовов функций библиотеки wxWidgets, их распараллеливание без исправления исходных текстов библиотеки не представляется возможным (что, однако, оказывается вмешательством во внутреннюю работу программы стороннего производителя и может привести к проблемам при работе).

Поэтому основная работа сосредоточена на распараллеливании основного цикла обработки изображения с использованием пробных боксов. Напомним, что получение мультифрактальных характеристик основано на линеаризации графика зависимости статистических характеристик изображения от размера пробного бокса, причем количество различных размеров пробных боксов достигает нескольких десятков (а то и превышает сотню, в зависимости от размера исходного изображения и заданных параметров просчета).

При этом очень удачным фактором, способствующим простоте распараллеливания вычислений, является то, что исходные данные для расчетов (пиксели изображения) используются только для чтения; никакой их модификации при вычислениях не производится. Таким образом, нет необходимости в защите исходных данных от состояния гонки с помощью взаимоблокировок, критических разделов или иных средств синхронизации потоков.

То же самое относится и к результатам расчетов. Теоретически все расчетные данные для разных размеров боксов сохраняются отдельно, так что вопрос о наличии состояния гонки, защиты обращения с помощью взаимоблокировок и прочего по сути не встает. Тем не менее определенная сложность здесь все же имеется. Поскольку результаты вычислений сохраняются в стандартном контейнере C++ — векторе `std::vector<>`, доступ к элементам которого выполняется через его функции-члены, более безопасным, эффективным и простым решением было сочтено не выполнять синхронизацию вызовов, а помещать результаты непосредственно в элементы заранее подготовленного вектора, доступ к которым осуществляется по конкретным, неизменным во время расчетов ссылкам на них.

Фактически основной распараллеливаемый цикл имеет следующий вид:

```
for(auto& e : expContainer)
{
    e.squareCount();
}
```

Здесь `expContainer` — подготовленный заранее вектор, элементами которого являются структуры, содержащие все необходимые для выполнения вычислений параметры — размер бокса, ссылку на исходные данные изображения, а также поля для размещения в них результатов расчетов.

Как демонстрирует приведенный ниже график, время расчета с достаточно высокой точностью имеет степенную зависимость от размера пробного бокса, выражающуюся как

$$t \sim s^{-0.61} \quad (1)$$

где $s$ — размер пробного бокса в пикселях, а $t$ — время расчета (вычисления функции-члена `squareCount()`).

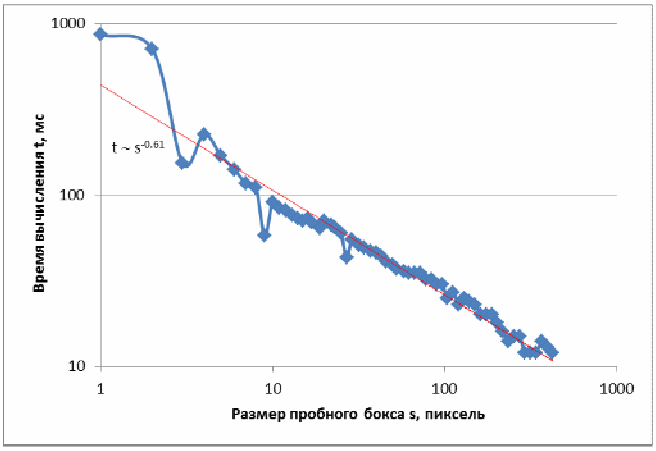


*Рисунок 1. Зависимость времени обработки пробных боксов от их размера*

Работа выполнялась на компьютере с четырехъядерным процессором Intel Core i5-2500 3.30 GHz, под управлением 64-разрядной операционной системы Windows 7. В качестве тестового изображения был выбран ковер Серпинского размером 6561×6561 пиксель. Среднее время выполнения основного цикла в однопоточном варианте составляло 4.7 с (при выполнении 97 итераций с разными размерами боксов).

### *1. Применение OpenMP*

До появления стандарта C++11 традиционно наиболее эффективным способом программирования многопоточности в Visual C++ считался поддерживаемый этим компилятором открытый стандарт OpenMP (хотя его применение оказывается достаточно сложной задачей, в особенности для неопытного программиста [11]).

При применении OpenMP, помимо описанных в работе [11] проблем, пришлось столкнуться с тем, что текущая реализация OpenMP в Visual C++ не умеет работать с современными конструкциями языка программирования C++, появившимися в стандарте C++11. Так, применение директивы

```
#pragma omp for
```

к циклу C++11 по диапазону

```
for(auto value: container)
```

воспринимается компилятором как ошибка. Это не является критичным препятствием для распараллеливания, но тем не менее оказывается достаточно неприятным фактором, требующим изменения исходного текста. Кроме того, пришлось создавать глобальную функцию (не являющуюся членом класса), и использовать ее — что существенно усложнило код, который приобрел следующий вид (для упрощения определение глобальной функции `squareCount()` и ряд других изменений кода опущены; опущены также директивы OpenMP):

```
int siz = expContainer.size();
int i;
for(i = 0; i < siz; ++i)
{
    experiment& e = expContainer[i];
    sqData sq = squareCount(e.size);
    e.data = sq.data;
}
```

Большим недостатком рассматриваемого метода является также невозможность автоматической обработки исключений (стандартного механизма обработки ошибок в C++), генерируемых в распараллеливаемом с помощью OpenMP коде. Такие исключения приходится обрабатывать непосредственно в месте генерации, например, с последующей записью соответствующей информации об ошибке в глобальные переменные, что, естественно, полностью лишает смысла стандартный способ обработки ошибок, не соответствует парадигме обработки ошибок в C++, и вносит дополнительные проблемы, связанные с конкурентной записью в память (требуя применения методов синхронизации). Фактически обработка ошибок сводится к выставлению флага и его проверке в основном потоке программы.

Эффективность непосредственного применения OpenMP оказалась достаточно низкой — время вычисления снизилось с указанных ранее 4.7 с до 3.7 с. Такая малая эффективность связана с тем, что наибольшее время вычисления, согласно (1), затрачивается на пробный бокс размером 1 пиксель. С ростом размера бокса время расчета полиномиально уменьшается, так что оптимальным является запуск потоков, начиная с наименьшего размера бокса. Это позволяет остальным потокам за то же время выполнить обработку боксов меньшего размера. К сожалению, непосредственное применение директив OpenMP для распараллеливания кода

```
#pragma omp parallel num_threads(4)
{
   #pragma omp for
   for(i = 0; i < siz; ++i)
   {
      experiment& e = expContainer[i];
```

где с ростом счетчика цикла `i` увеличивается размер бокса, приводит к тому, что первыми начинают выполняться потоки *не* для минимальных значений `i`, и более потому поздний запуск выполнения потока для единичного бокса практически устраняет весь эффект от распараллеливания вычислений. Для того же, чтобы заставить OpenMP запускать потоки в том порядке, который является наиболее эффективным, требуется прибегать к специальным мерам, существенно усложняющим код и без того усложненной по сравнению с однопоточной версией программы.

*2. Применение стандартных средств C++11*

Применение же стандартных средств библиотеки C++11 [12] оказывается более простым, чем применение OpenMP, но при этом лишенным его недостатков, в первую очередь связанных с обработкой исключений C++ и новыми возможностями стандарта C++11. Собственно, вся разница между нераспараллеленным кодом и кодом с параллельными вычислениями свелась к замене строки явного вызова функции расчета `squareCount()` к вызову стандартной функции `async<>()` (согласно рекомендации, приведенной в [13], предпочтительнее использовать параллельность на уровне задач, а не потоков) с передачей ей в качестве параметра функции `squareCount()`, сохранению возвращенного фьючерса и последующему вызову для фьючерсов функции-члена `get()`.

Таким образом, код

```
for(auto& e : expContainer)
{
    e.squareCount();
}
```

по существу заменяется кодом

```
for(auto& e : expContainer)
{
    e.f = async(
        &BFractal::squareCount,this);
}
```

с последующим циклом ожидания завершения расчетов

```
for(auto& e : exps) { e.f.get(); }
```

Никаких дополнительных действий (определения глобальных функций и т.п.) помимо добавления поля для фьючерса в структуру данных, описывающих вычислительный эксперимент, применение данного метода не требует.

Указанные изменения в коде приводят к асинхронному (по возможности) выполнению переданной функции-члена. При этом в наихудшем случае невозможности параллельного (асинхронного) выполнения задач все они будут выполнены последовательно (синхронно), т.е. никакого ухудшения производительности при невозможности запуска в многопоточном режиме по сравнению с исходной, однопоточной программой практически не будет.

В результате применения функции асинхронного выполнения `async<>()` время расчета снизилось до 1.3 с, т.е. в 3,6 раза, что достаточно близко к теоретическому выигрышу в 4 раза (несоответствие максимально допустимому выигрышу объясняется тем, что обеспечить оптимальное распределение задач по потокам при сложной зависимости времени вычислений (1) практически нереально).

Регулировка запуска задач в требуемом для высокой эффективности порядке оказалась при данном подходе по сути ненужной (программа хорошо работает и без нее; если бы такая настройка и понадобилась, то с помощью параметра стратегии запуска функции `async<>()` ее было бы легко осуществить).

*3. Применение средств Windows API*

Применение средств Windows API дало практически ту же эффективность, что и применение стандартных средств языка программирования C++, но при этом оно обладает массой недостатков, в первую очередь — сложным написанием функции запуска потоков, необходимостью обеспечения специальной передачи вычисленных результатов из потока, невозможностью простой и естественной обработки исключений C++.

Главным преимуществом применения низкоуровневого Windows API является возможность тонкой настройки работы потоков — например, их приоритета или иных действий, но: 1) в вычислительных задачах это требуется достаточно редко, а в нашей — по крайней мере в случае применения стандартных средств C++ — не требуется вовсе; 2) при необходимости такой тонкой настройки мы могли бы использовать другое средство стандартной библиотеки C++ для параллельный вычислений — `thread<>()`, которое обеспечивает указанные возможности посредством получения дескриптора потока и его использования в функциях Windows API. (Заметим, однако, что цена применения более низкоуровневого средства `thread<>()` оказывается выше, чем применения `async<>()` — в частности, в силу той же невозможности централизованной обработки исключений.)

## III. Заключение

В результате проведенной работы получены следующие времена выполнения цикла для тестового изображения:

| Однопоточное | OpenMP | C++11 | Windows API |
|---|---|---|---|
| 4.7 с | 3.7 с | 1.3 с | 1.3 с |

Что касается трудозатрат для использования различных средств распараллеливания, то оценить их можно только качественно, — от наибольших трудозатрат при применении низкоуровневых средств Windows API до наименьших при использовании стандартны средств стандарта C++11.

Подводя итог, констатируем, что при распараллеливании вычислений наилучшую эффективность при максимальной простоте применения и соответствии парадигмам программирования языка C++ дает использование стандартных средств C++11.

Показана также оправданность применения распараллеливания в программе вычисления мультифрактальных характеристик двумерных изображений.

## Библиография


1. Встовский Г.В. *Элементы информационной физики*. М.:МГИУ, 2002. — 260 с.
2. Иванова В.С., Встовский Г.В., Колмаков А.Г., Пименов В.Н. *Мультифрактальный метод тестирования устойчивости структур в материалах*. — М.: Интерконтакт Наука, 2000, 54 с.
3. *И.Е. Красикова, И.В. Красиков, В.В. Картузов*. Определение фрактальных характеристик структуры материалов методом мультифрактального анализа изображений. Вычислительный эксперимент на модельных объектах. // Математические модели и вычислительный эксперимент в материаловедении. — К.: Ин-т пробл. материаловедения им. И.Н. Францевича НАН Украины. — 2007. — Вып. 9. — С.79–84.
4. *И.Е. Красикова, В.В. Картузов, И.В. Красиков*. Компьютерная реализация алгоритма вычисления фрактальной размерности структуры материала по изображениям, полученным при помощи электронной микроскопии. // Математические модели и вычислительный эксперимент в материаловедении. — К.: Ин-т пробл. материаловедения им. И.Н. Францевича НАН Украины. — 2011. — Вып. 13. — С.82–89.
5. *И.Е. Красикова, В.В. Картузов, И.В. Красиков*. Компьютерная реализация алгоритма вычисления мультифрактальных характеристик структуры материала по двумерным изображениям. // Математические модели и вычислительный эксперимент в материаловедении. — К.: Ин-т пробл. материаловедения им. И.Н. Францевича НАН Украины. — 2014. — Вып. 16. — С.74–79.
6. *И.Е. Красикова, В.В. Картузов, И.В. Красиков*. Характеристики компьютерной реализации алгоритма вычисления фрактальной размерности двумерных изображений. // Математические модели и вычислительный эксперимент в материаловедении. — К.: Ин-т пробл. материаловедения им. И.Н. Францевича НАН Украины. — 2013. — Вып. 15. — С.69–73.
7. Закон Мура. — https://ru.wikipedia.org/wiki/Закон_Мура
8. *Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн*. Алгоритмы: построение и анализ, 3-е изд. — М.: Издательский дом “Вильямс”, 2013 г. — 1328 с.
9. https://ru.wikipedia.org/wiki/OpenMP
10. *Э. Уильямс*. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. — М.: ДМК Пресс, 2012. — 672 с.
11. *А. Колосов, А. Карпов, Е. Рыжков*. 32 подводных камня OpenMP при программировании на Си++. — http://www.viva64.com/ru/a/0054/
12. *Н. М. Джосаттис*. Стандартная библиотека C++: справочное руководство, 2-е изд. — М.:ООО “И.Д. Вильямс”, 2014. — 1136 с.
13. *S. Meyers*. Effective Modern C++. — O’Reilly Media, Inc., 2015.

# Using computation parallelization facilities in the 2D-image fractal dimension evaluation algorithm

I.E. Krasikova, V.V. Kartuzov, I.V. Krasikov

*Abstract* — **The paper deals with the application of parallel computing facilities in the C++ programming language (Windows API, OpenMP, C++11) to the computer implementation of the algorithm for calculating 2D-images fractal dimension. The advantages and disadvantages of these methods are considered. It's shown that the simplest and most effective method is using of C++11 standard features.**

*Keywords* — **algorithm, fractal dimension, parallel computing, OpenMP, C++11, dynamical multithreading, multiprocessor systems**