Лабораторная работа 1 – Разработка нового распределенного приложения для инструментария Alchemi

1. Введение

В теоретической части курса, посвященной инструментарию Alchemi, была рассмотрена архитектурная схема данного инструментария, принцип его работы, возможные сценарии применения и некоторые вопросы, связанные с разработкой приложений для грид. Цикл лабораторных работ ставит своей целью практическое знакомство с инструментарием Alchemi и основными его возможностями.

Напомним, что данный инструментарий предоставляет две модели программирования:

- модель *грид-потоков*,
- модель *грид-заданий* на основе файлов.

Первая модель программирования – модель грид-потоков – предназначена, прежде всего, для разработки новых приложений для грид, построенной на базе Alchemi. Это основной способ разработки приложений, уникальный для данного инструментария. Модель грид-заданий на основе файлов предусмотрена для обеспечения совместимости с другим программным обеспечением промежуточного уровня. Благодаря этому вычислительная грид, построенная на базе Alchemi, может входить в состав более крупной грид, построенной на базе "традиционного" программного обеспечения промежуточного уровня (например, на базе пакета Globus Toolkit).

Таким образом, основной интерес для изучения представляет модель грид-потоков, так как именно она использует все возможности инструментария Alchemi и платформы Microsoft .NET Framework. Данная модель программирования является весьма эффективной и гибкой. Как уже было сказано, с помощью модели грид-потоков можно довольно *просто* и *быстро* разработать *новое* приложение для грид. Однако в ряде случаев в рамках данной модели возможно внедрение в грид *существующего* приложения. В последнем случае с помощью инструментария Alchemi разрабатывается некоторая сервисная программа для запуска существующего приложения на нескольких узлах, а также для отображения итогового результата. Обсуждение этих двух аспектов применения модели грид-потоков является основной задачей лабораторных работ по инструментарию Alchemi.

В данной лабораторной работе подробно рассматривается процесс разработки *нового* приложения для грид на примере задачи визуализации функций комплексного переменного. Во все времена математики искали простые и наглядные образы, позволяющие лучше понять суть задачи. Для изучения функций мы постоянно прибегаем к графикам, которые позволяют наглядно судить об их поведении. Однако как визуально представить столь интересный математический объект, как функция комплексного переменного? Хотелось бы иметь простой и наглядный способ визуализации таких функций, который бы облегчал задачу анализа так же, как график обычной действительной функции. Интересно, что существует довольно много таких методов, и у каждого из них есть свои достоинства и недостатки. В данной лабораторной работе рассматривается один из таких методов, который будет реализован в виде программы для инструментария Alchemi.

2. Цель лабораторной работы

Целью лабораторной работы является разработка *нового* распределенного приложения для инструментария Alchemi, решающего задачу визуализации функций комплексного переменного описанным ниже методом.

- *Упражнение 1*.

  Постановка задачи визуализации функции комплексного переменного.
- *Упражнение 2*.

  Разработка алгоритма визуализации функции комплексного переменного.
- *Упражнение 3*.

  Декомпозиция вычислительной схемы на независимые части.
- *Упражнение 4*.

  Разработка распределенного приложения для инструментария Alchemi.
- *Упражнение 5*.

  Получение сведений о выполняемом распределенном приложении.

Примерное время выполнения лабораторной работы: **240 минут**.

При выполнении лабораторной работы предполагаются знания раздела "Инструментарий Alchemi" курса по технологиям грид, а также базовые знания языка программирования Visual C# и навыки работы в среде Microsoft Visual Studio.

3. Упражнение 1 – Постановка задачи визуализации функции комплексного переменного

Напомним, что каждое *комплексное число* представляет собой пару действительных чисел (x, y), где x - *действительная* часть комплексного числа, а y - его *мнимая* часть. Поэтому естественно воспринимать комплексное число как двумерный вектор с координатами x и y. *Функцией* комплексного переменного $w = f(z)$ называется правило, посредством которого всякому комплексному числу z из некоторого множества D ставится в соответствие комплексное число w из множества W. Множество D называется *областью определения* функции $f(z)$, а множество W – ее *областью значений*.

Итак, пусть рассматривается некоторая функция комплексного переменного $w = f(z)$, заданная на прямоугольнике $D = [a, b] \times [c, d]$. Исходной информацией для задачи являются следующие данные:

- символьное представление данной функции,
- границы прямоугольника D, в котором будет проводиться визуализация.

Будем предполагать, что на заданном прямоугольнике D функция $w = f(z)$ является однозначной. В противном случае будем рассматривать только главную ветвь функции, все остальные ветви можно получить с помощью преобразований уравнения. Задача состоит в том, чтобы построить геометрический образ заданной функции в области D.

Можно предложить множество различных способов визуализации функций комплексного переменного. Для решения данной задачи предлагается воспользоваться описанным ниже методом.

4. Упражнение 2 – Разработка алгоритма визуализации функций комплексного переменного

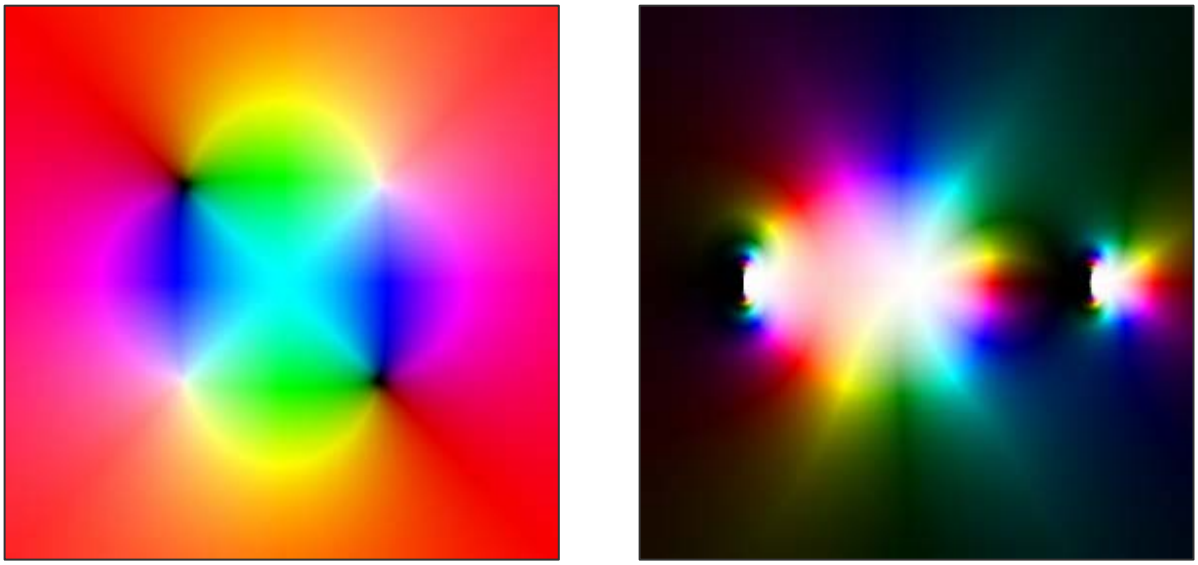
График комплексной функции одного переменного лежит в двумерном комплексном пространстве, которое может быть представлено как обычное четырехмерное действительное пространство. Это означает, что если $z = x + iy$ и $w = u + iv$, то точку $(z, w(z))$ графика функции $w = f(z)$ можно мыслить как точку четырехмерного пространства (x, y, u, v). Тогда график есть поверхность в четырехмерном пространстве. Как мы можем изобразить эту поверхность? Очевидно, что аргумент функции – комплексное число z – можно изобразить как точку на плоскости с координатами (x, y). Значение же w функции $w = f(z)$ можно изобразить, например, с помощью цвета. Для этого каждому комплексному числу нужно поставить в соответствие *уникальный* цвет.

а) *б)* *в)*

Рис. 1. Результат сопоставления каждому комплексному числу уникального цвета

Комплексному числу 1 сопоставим *красный* цвет, двум другим кубичным корням из единицы – $e^{2\pi i/3}$ и $e^{-2\pi i/3}$ – *зеленый* и *синий* соответственно. Для определения цвета, соответствующего всем остальным значениям аргумента, воспользуемся линейной интерполяцией. Рассмотрим, например, точку единичной окружности $e^{\pi i/3}$, которая имеет аргумент $\pi/3$. Поскольку значение $\pi/3$ является серединой отрезка $[0, 2\pi/3]$, концам которого соответствуют красный и зеленый цвет, то для получения цвета данной точки указанные цвета необходимо смешать в равных пропорциях. Таким образом, каждый луч на комплексной плоскости с вершиной в нуле (и, стало быть, каждый аргумент комплексного числа) ассоциируется с некоторым уникальным цветом (рис. 1*а*). Для того чтобы учесть модуль комплексного числа, ноль окрасим белым цветом, а бесконечно удаленную точку – черным. Тогда любое значение модуля комплексного числа получит уникальный оттенок серого цвета (рис. 1*б*). Объединяя эти два правила, для каждого комплексного числа (то есть для каждой пары из модуля и аргумента) получаем уникальный цвет (рис. 1*в*). Приведенных пояснений в целом должно быть достаточно для качественного понимания алгоритма назначения цветов и успешного выполнения данной лабораторной работы. Для

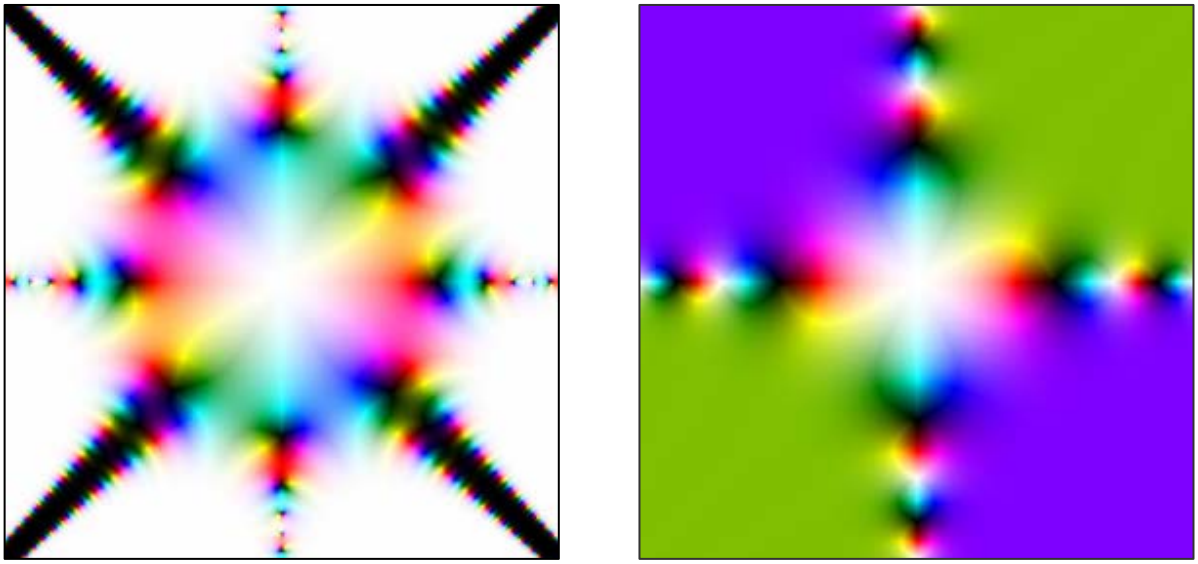
более подробного изучения алгоритма можно обратиться к исходным кодам соответствующей функции (рассматривается далее).



а) *б)*

Рис. 2. Графики комплексных функций $w = (z^2 - i) / (z^2 + i)$ и $w = z^2 \cdot e^{tg(z)}$

Далее в этом упражнении в качестве иллюстрации описанного метода мы рассмотрим некоторые комплексные функции и проведем их простейшее исследование. Данный материал не является обязательным для прочтения и может быть легко пропущен.



а) *б)*

Рис. 3. Графики комплексных функций $w = sin(z^2) / cos(z^4)$ и $w = tg(z^2)$

Изобразим этим методом некоторые комплексные функции. На рис. 2*а* представлена функция $w = (z^2 - i) / (z^2 + i)$. Для нее точки $z = e^{i\pi/4}$ и $z = e^{-i3\pi/4}$ являются нулями первого порядка. Именно поэтому они окрашены на графике в белый цвет. Точки $z = e^{-i\pi/4}$ и $z = e^{i3\pi/4}$ являются полюсами первого порядка, поэтому они окрашены в черный цвет. Следует заметить, что по графику функции можно определить даже порядок нулей и полюсов. Для этого нужно посчитать, сколько раз встречается каждый опорный цвет (красный, зеленый или синий) при обходе вокруг этих точек. В нашем случае каждый цвет встречается лишь один раз, что и подтверждается теорией. Таким способом функции комплексного переменного могут быть исследованы даже в окрестности существенно особых точек. На рис. 2*б* приведена функция $w = z^2 \cdot e^{tg(z)}$. Для нее существенно особыми точками будут служить полюса функции $w = tg(z)$, т. е. точки $z = \pi/2 + \pi k$. Кроме того, точка $z = 0$ является нулем второго порядка для нашей функции, поэтому она окрашена белым цветом и при обходе вокруг данной точки цвета повторяются

дважды. На рис. 3 изображены функции $w = sin(z^2) / cos(z^4)$ и $w = tg(z^2)$. Для них также легко проделать описанные выше рассуждения.

Описанный метод является досаточно эффективным при изучении комплексных функций. Глядя на график, о функции можно сказать довольно много: определить особые точки и нули, установить кратность нулей и полюсов, определить области однолистности, охарактеризовать общее поведение функции.

Визуализация больших изображений для сложных функций является трудоемкой вычислительной задачей. Для ее решения воспользуемся инструментарием Alchemi.

5. Упражнение 3 – Декомпозиция вычислительной схемы на независимые части

Для решения некоторой вычислительной задачи с помощью инструментария Alchemi необходимо проделать следующие основные шаги:

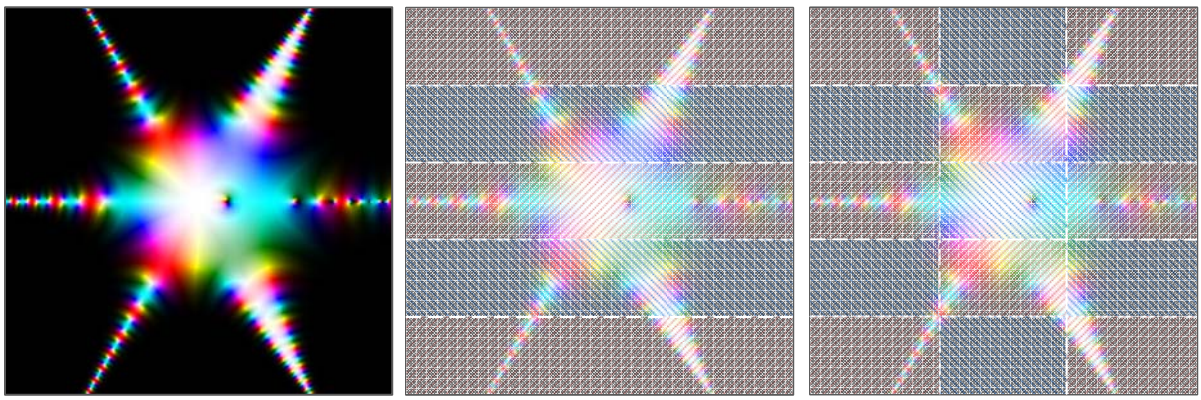
- произвести декомпозицию вычислительной схемы на *независимые* подзадачи;
- реализовать *грид-потоки* для обработки данных подзадач;
- организовать выполнение грид-потоков в вычислительной сети с помощью класса *грид-приложения*.

В этом упражнении мы кратко обсудим первый этап разработки приложения. Выбор способа разделения вычислений на независимые части основывается на анализе вычислительной схемы решения задачи. При использовании инструментария Alchemi можно выделить *два* основных требования, которым должен удовлетворять выбираемый подход:

- обеспечение примерно равного объема вычислений в выделяемых подзадачах;
- отсутствие информационных зависимостей между этими подзадачами.

Поскольку в общем случае проведение анализа и выделение задач представляет собой достаточно сложную проблему, рассмотрим два часто встречающихся типа вычислительных схем.

Для большого класса задач вычисления сводятся к выполнению однотипной обработки элементов большого набора данных. В этом случае говорят, что существует *параллелизм по данным*, и выделение подзадач сводится к разделению имеющихся данных. Для нашей учебной задачи исходное множество пикселей генерируемого изображения может быть разделено на отдельные группы строк – *ленточная* схема разделения данных или на прямоугольные наборы пикселей – *блочная* схема разделения данных (рис. 4).



а) *б)* *в)*

Рис. 4. Варианты разделения множества пикселей изображения: исходное изображение (*а*), ленточная схема (*б*) и блочная схема (*в*)

Для другой части задач вычисления могут состоять в выполнении разных операций над одним и тем же набором данных – в этом случае говорят о существовании *функционального параллелизма*. Предположим, например, что нам требовалось бы визуализировать комплексную функцию несколькими различными методами. Очевидно, что входной набор данных будет одинаковым для каждого такого метода. При этом расчеты можно производить совершенно независимо на различных вычислительных узлах.

Заметим, что в ряде случаев параллелизм по данным и функциональный параллелизм удобно сочетать. Так, например, при визуализации комплексной функции различными методами исходную

область можно предварительно разбить на части и обрабатывать их совершенно независимо, применяя при этом различные алгоритмы.

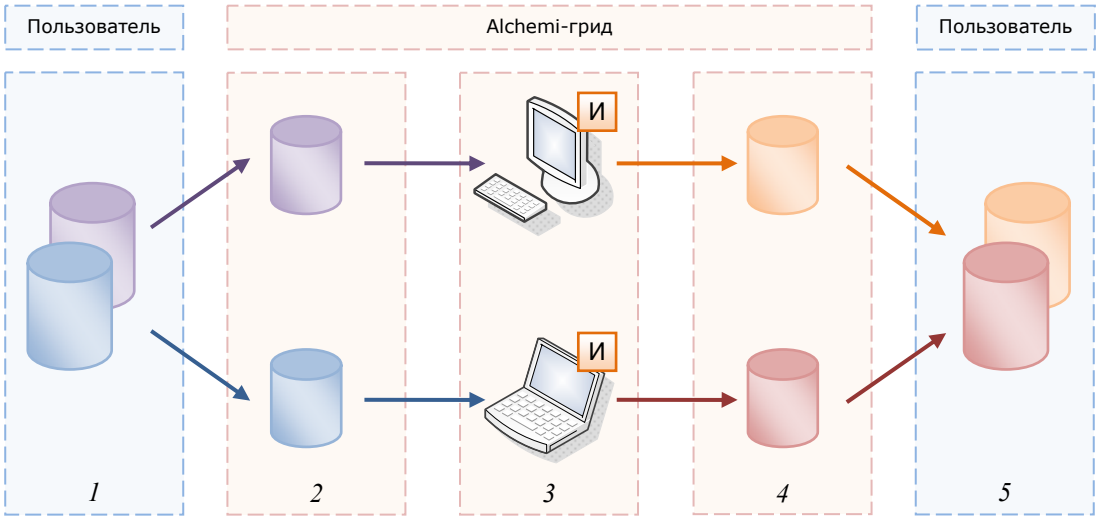


Рис. 5. Процесс распределенной обработки данных в виде последовательности пяти основных шагов

Общая схема распределенных вычислений с использованием инструментария Alchemi представлена на рис. 5. На *первом* шаге производится декомпозиция входного набора данных на независимые части, при этом используется параллелизм по данным рассматриваемой задачи. На *втором* шаге отдельные порции данных загружаются на машины Исполнителей для дальнейшей обработки. На *третьем* шаге производится обработка порций данных независимыми Исполнителями, при этом отдельные порции могут обрабатываться различным образом. Именно на этом шаге используется функциональный параллелизм. В результате на *четвертом* шаге получаются частичные результаты обработки (на удаленных узлах), которые впоследствии на *пятом* шаге объединяются в итоговый результат (на компьютере пользователя).

Для рассматриваемого учебного примера мы ограничимся использованием параллелизма по данным с блочной схемой разбиения изображения. Однако при разработке практических программ рекомендуется внимательно изучить вычислительную схему на наличие функционального параллелизма и параллелизма по данным для выделения максимального количества независимых подзадач. Такой подход позволит получить наибольшую отдачу от имеющихся ресурсов.

В заключение заметим, что при декомпозиции вычислительной схемы всегда нужно учитывать накладные расходы на установку соединения и общее число имеющихся Исполнителей. Объем вычислений, производимый одним грид-потоком, не должен быть слишком мал, иначе не окупятся расходы, связанные с его передачей на удаленную машину. Общее число грид-потоков не должно в десятки или даже сотни раз превышать число доступных Исполнителей. В этом случае несколько грид-потоков лучше объединить в один более крупный (например, обрабатывать в каждом грид-потоке более крупные порции данных).

К сожалению, для времени передачи грид-потоков крайне трудно дать общую оценку. Данный параметр существенно зависит от имеющейся в распоряжении сети, скорости сетевого соединения, рабочих станций и даже версии используемого программного обеспечения (например, в последней на момент написания документа версии инструментария Alchemi (1.0.6) разработчики значительно сократили время передачи грид-потоков). Однако следует учитывать, что временные расходы на передачу грид-потоков могут составить *несколько секунд*. Отчасти столь существенные временные задержки связаны со следующим обстоятельством. Между пользовательским компьютером, на котором запущено грид-приложение, и Менеджером устанавливается лишь односторонняя связь (от машины пользователя к Менеджеру). Именно это обстоятельство позволяет грид-приложению успешно работать даже при ограничениях сети. Однако это влечет и обратный эффект. Завершенные на удаленных Исполнителях грид-потоки отправляются Менеджеру, который не может отправить их незамедлительно на узел пользователя. Грид-потоки хранятся на Менеджере до тех пор, пока грид-приложение не загрузит их самостоятельно, при этом обращения к Менеджеру для проверки наличия завершенных грид-потоков осуществляются с некоторой периодичностью (в последней версии период составляет 700 миллисекунд). Таким образом, даже наиболее “быстрые” грид-потоки не удастся получить ранее установленного периода. Заметим также, что на первый взгляд выход из этой проблемы очевиден – увеличить частоту обращения к Менеджеру. Однако этого нельзя допускать, поскольку сетевое соединение будет

использоваться неразумно, обслуживая преимущественно “пустые” запросы. В будущем ситуация может измениться, поэтому для получения более точных оценок времени передачи грид-потоков его необходимо измерять для каждого конкретного случая. Для этого, например, можно запустить “пустые” грид-потоки (которые не производят никаких вычислений), и проследить, через какое время они будут завершены. Дополнительная информация по данному аспекту организации распределенных вычислений может быть получена, например, в разделе 3 учебного курса “Теория и практика параллельного программирования для систем с распределенной памятью”.

6. *Упражнение 4 – Разработка распределенного приложения*

После того, как завершена декомпозиция вычислительной схемы на независимые части, можно приступать к реализации распределенного приложения. Начальный вариант будущей программы представлен в проекте **ComplexVisual**, который содержит некоторую часть исходного кода. В ходе выполнения дальнейших упражнений необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, реализацией описанного алгоритма и проверкой правильности результатов работы программы.

6.1. Задание 1 – Открытие проекта ComplexVisual

Для открытия проекта **ComplexVisual** выполните следующие шаги:

- Запустите среду Microsoft Visual Studio 2005, если она еще не запущена,
- В меню **File** выполните команду **Open | Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **ComplexVisual**,
- Выберите файл **ComplexVisual.sln** и выполните команду **Open**.

После выполнения описанных шагов в окне **Solution Explorer** (рис. 6) будет отображена структура проекта **ComplexVisual**. В его состав входит приложение **AlchemiApplication**, а также две динамические библиотеки **ComplexThread** и **MathTools**. Опишем их назначение подробнее.

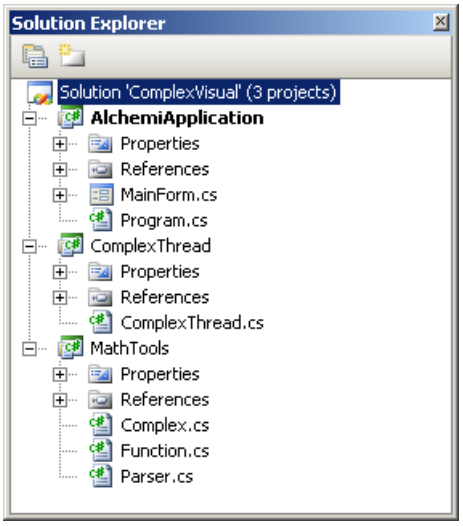


Рис. 6. Структура проекта **ComplexVisual**

Динамическая библиотека **MathTools** содержит вспомогательные математические классы, которые используются основным приложением. Несмотря на то, что данная библиотека является вспомогательной и не требует дальнейшей доработки, кратко рассмотрим ее состав. Обзор библиотеки **MathTools** не является обязательным для данной лабораторной работы и может быть легко пропущен. Заметим, однако, что реализованные в данной библиотеке классы могут быть использованы и при разработке ряда других приложений.

В файле **Complex.cs** определяется класс комплексных чисел **`Complex`**, основные операции над ними, а также базовые функции комплексных переменных. К числу таких операций относятся операции сложения, вычитания, умножения и деления комплексных чисел, а также операция унарный минус. К числу базовых функций, реализованных в данном классе, относятся функции $w = sin(z)$, $w = cos(z)$, $w = log(z)$, $w = exp(z)$, $w = abs(z)$, $w = arg(z)$ и др. В качестве примера использования класса **`Complex`**, создадим несколько комплексных переменных и произведем над ними некоторые операции:

```
using MathTools;

...

// Создадим два комплексных числа a и b
Complex a = new Complex(1, 5);
Complex b = new Complex(5, 8);

// Произведем некоторые арифметические операции
Complex c = a * b + a / b;

// Вычислим некоторые комплексные функции
Complex w = Complex.Sin(Complex.Log(c - Complex.Sqrt(c)));
```

В файле **Function.cs** определяется базовый класс для всех комплексных функций **`BaseFunction`**, а также базовые операции над функциями, такие как операции сложения, вычитания, умножения, деления, а также операция унарный минус. От базового класса **`BaseFunction`** наследованы классы конкретных комплексных функций: **`SinFunction`**, **`CosFunction`**, **`LogFunction`**, **`ExpFunction`**, **`SqrtFunction`** и т. д. При вычислении этих функций используются методы класса **`Complex`**. На приведенном ниже листинге показано объявление базового *абстрактного* класса **`BaseFunction`** и нескольких производных от него классов. Все остальные производные классы определяются совершенно аналогично.

```
[Serializable]
public abstract class BaseFunction
{
    // Вычисляет значение комплексной функции
    public abstract Complex CalcFunction(Complex arg);

    // Вычисляет сумму двух комплексных функций
    public static BaseFunction operator +(BaseFunction left,
                                           BaseFunction right)
    {
        return new AddFunction(left, right);
    }

    // Вычисляет разность двух комплексных функций
    public static BaseFunction operator -(BaseFunction left,
                                          BaseFunction right)
    {
        return new SubFunction(left, right);
    }

    // Вычисляет произведение двух комплексных функций
    public static BaseFunction operator *(BaseFunction left,
                                          BaseFunction right)
    {
        return new MulFunction(left, right);
    }

    // Вычисляет частное двух комплексных функций
    public static BaseFunction operator /(BaseFunction left,
                                          BaseFunction right)
    {
        return new DivFunction(left, right);
    }

    // Вычисляет комплексную функцию, противоположную к данной
    public static BaseFunction operator -(BaseFunction funct)
    {
        return new NegativeFunction(funct);
    }
}

[Serializable]
public class AddFunction : BaseFunction
{
    private BaseFunction left;
    private BaseFunction right;

    public AddFunction(BaseFunction left, BaseFunction right)
    {
        this.left = left;
        this.right = right;
    }

    public override Complex CalcFunction(Complex arg)
    {
        return left.CalcFunction(arg) + right.CalcFunction(arg);
```

```
    }
}

[Serializable]
public class SinFunction : BaseFunction
{
    private BaseFunction funct;

    public SinFunction(BaseFunction funct)
    {
        this.funct = funct;
    }

    public override Complex CalcFunction(Complex arg)
    {
        return Complex.Sin(funct.CalcFunction(arg));
    }
}
```

С помощью данных классов можно организовывать иерархические структуры вложенных функций и вычислять их значение в произвольной точке, используя следующий метод абстрактного класса **`BaseFunction`**:

```
// Вычисляет значение комплексной функции
public abstract Complex CalcFunction(Complex arg);
```

В каждом производном классе данный метод имеет определенную реализацию. Сформируем, в качестве примера, комплексную функцию $w = sin(z) + cos(z)$ и вычислим ее значение в некоторых точках.

```
// Создадим тождественную функцию w = z
BaseFunction equal = new EqualFunction();

// Создадим функцию w = sin(z)
BaseFunction sin = new SinFunction(equal);

// Создадим функцию w = cos(z)
BaseFunction cos = new CosFunction(equal);

// Создадим функцию w = sin(z) + cos(z)
BaseFunction funct = new AddFunction(sin, cos);

// Создадим комплексный аргумент
Complex c = new Complex(1, 5);

// Вычислим значение функции в некоторых точках
Complex w = funct.CalcFunction(c);
Complex u = funct.CalcFunction(w + c);
```

Описанным выше способом можно сформировать любую комплексную функцию (из определенных выше элементарных функций) и вычислять ее значение. Однако определение функции предполагает ее реализацию. Для упрощения задания комплексных функций можно обеспечить, например, возможность их формульного (аналитического) описания в виде обычных текстовых строк. Для обеспечения такой возможности библиотека **MathTools** содержит модуль синтаксического анализа.

В файле **Parser.cs** определяется класс **`Parser`** для синтаксического анализа строки и формирования на ее основе последовательности вложенных функций. В процессе разбора формируется древовидная структура функций, на основе которой вычисляется значение введенной функции без повторного анализа строки, что значительно повышает производительность. Здесь не приводится дополнительная информация по синтаксическому анализу и различным его методам. Более подробную информацию можно найти, например, в [1]. Для того чтобы прояснить использование модуля синтаксического анализа, рассмотрим конкретный пример. Пусть требуется вычислить значение комплексной функции в некоторой точке z, заданной при помощи строки "$sin(cos(z)) + exp(z)$". Для этого, прежде всего, нужно произвести синтаксический анализ входной строки при помощи статического метода класса **`Parser`**:

```
// Формируем суперпозицию вложенных функций
BaseFunction funct = Parser.ParseString("sin(cos(z))+exp(z)");

// Проверяем, удалось ли сформировать функцию
if (funct != null)
{
    // Вычисляем значение функции в любой точке
    Complex value = funct.CalcFunction(z);
}
```

Суперпозиция вложенных функций, сформированная на основе указанной выше входной строки, будет выглядеть так, как показано на рис. 7.

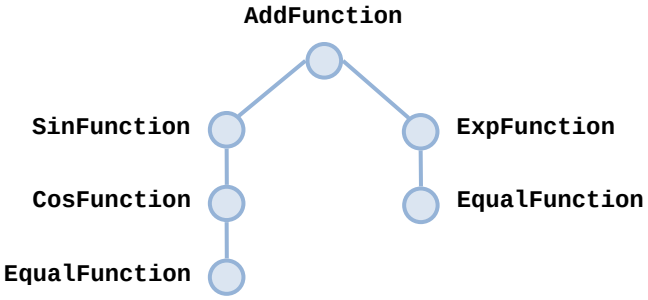


Рис. 7. Древовидная структура функций, сформированная на основе входной строки "*sin*(*cos*(*z*)) + *exp*(*z*)"

Для более подробного изучения алгоритма синтаксического анализа и вычисления значения функции можно обратиться к исходным кодам библиотеки **`MathTools`**. Данная библиотека является примером использования объектно-ориентированного подхода для реализации синтаксического анализа математических выражений. Этот же алгоритм может быть использован для реализации символьного дифференцирования функций, а также в ряде других случаев.

Все классы, объявленные в библиотеке **`MathTools`**, потенциально могут быть отправлены через сеть и потому имеют атрибут **`Serializable`**, который снабжает класс специальными методами *сериализации* (*serialization*) и *десериализации* (*deserialization*). В процессе сериализации состояние объекта преобразуется в форму, которая может быть сохранена в файл или передана по сети. Обратным процессом, при котором некоторый поток данных преобразуется в объект, является десериализация. В совокупности процессы сериализации и десериализации позволяют легко сохранять объекты в файлы (и загружать из файлов) или передавать их по сети.

Перейдем к рассмотрению проектов **AlchemiApplication** и **ComplexThread**. Оба проекта подлежат дальнейшей доработке и содержат для этого необходимые комментарии.

Динамическая библиотека **ComplexThread** состоит из единственного файла **ComplexThread.cs**, в котором объявляется класс грид-потока **`ComplexThread`**. Напомним, что класс грид-потока *всегда* пересылается по сети на машины Исполнителей. Поэтому перед запуском приложения на выполнение инструментарий Alchemi копирует на машины Исполнителей модуль, в котором объявляется данный класс, а также все вспомогательные модули, которые используются в процессе работы грид-потока (в нашем случае это динамическая библиотека **MathTools**). Если класс грид-потока объявить в проекте основного приложения, то на машины Исполнителей при каждом запуске вычислений будет пересылаться вся программа целиком. Это особенно нежелательно при больших размерах приложения. Чтобы избежать такого поведения *класс грид-потока всегда следует реализовывать в виде отдельной динамической библиотеки*. Это же замечание справедливо в отношении других вспомогательных классов. Такой простой прием позволяет значительно сократить сетевой трафик и ускорить время начала вычислений, что может быть особенно важно для интерактивных программ.

Проект **AlchemiApplication** представляет собой обычное приложение для операционной системы Microsoft Windows. В данном приложении формируются грид-потоки и отправляются для исполнения в вычислительную грид, построенную на базе инструментария Alchemi. Кроме того, предусмотрена также возможность обработки грид-потоков на локальной машине пользователя. В дальнейших заданиях мы подробно рассмотрим реализацию библиотеки **ComplexThread** и приложения **AlchemiApplication**.

6.2. Задание 2 – Разработка класса грид-потока

Описав структуру проекта **ComplexVisual** и произведя декомпозицию вычислительной схемы на независимые части, перейдем к реализации класса грид-потока. Напомним, что в нашем учебном примере отдельные грид-потоки должны обрабатывать блоки пикселей изображения. На приведенном ниже листинге показана заготовка для класса грид-потока **`ComplexThread`**.

```
using System;
using System.Drawing;
using Alchemi.Core.Owner;
using MathTools;

/// <summary>
/// Грид-поток для вычисления отдельной части графика комплексной функции.
/// </summary>
[Serializable]
```

```
public class ComplexThread : GThread
{
    /// <summary>Ширина обрабатываемой части изображения в пикселях.</summary>
    private int width;

    /// <summary>Высота обрабатываемой части изображения в пикселях.</summary>
    private int height;

    /// <summary>Горизонтальный номер обрабатываемой части изображения.</summary>
    private int hornumber;

    /// <summary>Вертикальный номер обрабатываемой части изображения.</summary>
    private int vernumber;

    /// <summary>Минимальное значение аргумента x в обрабатываемой части области.</summary>
    private double xmin;

    /// <summary>Максимальное значение аргумента x в обрабатываемой части области.</summary>
    private double xmax;

    /// <summary>Минимальное значение аргумента y в обрабатываемой части области.</summary>
    private double ymin;

    /// <summary>Максимальное значение аргумента y в обрабатываемой части области.</summary>
    private double ymax;

    /// <summary>Комплексная функция для визуализации.</summary>
    private BaseFunction function;

    /// <summary>Насыщенность черного цвета в генерирумом изображении.</summary>
    private double white;

    /// <summary>Насыщенность белого цвета в генерирумом изображении.</summary>
    private double black;

    /// <summary>Точечный рисунок для хранения сгенерированной части изображения.</summary>
    private Bitmap image;

    /// <summary>
    /// Создает новый грид-поток для вычисления части графика.
    /// </summary>
    /// <param name="width">ширина части изображения</param>
    /// <param name="height">высота части изображения</param>
    /// <param name="hornumber">горизонтальный номер части изображения</param>
    /// <param name="vernumber">вертикальный номер части изображения</param>
    /// <param name="xmin">минимальное значение аргумента x в обрабатываемой области</param>
    /// <param name="xmax">максимальное значение аргумента x в обрабатываемой области</param>
    /// <param name="ymin">минимальное значение аргумента y в обрабатываемой области</param>
    /// <param name="ymax">максимальное значение аргумента y в обрабатываемой области</param>
    /// <param name="function">комплексная функция для визуализации</param>
    /// <param name="white">насыщенность белого цвета</param>
    /// <param name="black">насыщенность черного цвета</param>
    public ComplexThread(int width, int height, int hornumber, int vernumber,
                         double xmin, double xmax, double ymin, double ymax,
                         BaseFunction function, double white, double black)
    {
        ...
    }

    /// <summary>
    /// Основной метод. Обрабатывает часть графика комплексной функции.
    /// </summary>
    public override void Start()
    {
        ...
    }

    /// <summary>
    /// Вспомогательная функция. Определяет цвет, соответствующий
    /// заданному комплексному числу.
    /// </summary>
    /// <param name="value">комплексное число</param>
    /// <returns>цвет, соответствующий комплексному числу</returns>
    private Color CalcColor(Complex value)
    {
        ...
    }

    /// <summary>
```

```
    /// Вертикальный номер обрабатываемой части изображения.
    /// </summary>
    public int VerNumber
    {
        get
        {
            return vernumber;
        }
    }

    /// <summary>
    /// Горизонтальный номер обрабатываемой части изображения.
    /// </summary>
    public int HorNumber
    {
        get
        {
            return hornumber;
        }
    }

    /// <summary>
    /// Ширина обрабатываемой части изображения.
    /// </summary>
    public int Width
    {
        get
        {
            return width;
        }
    }

    /// <summary>
    /// Высота обрабатываемой части изображения.
    /// </summary>
    public int Height
    {
        get
        {
            return height;
        }
    }

    /// <summary>
    /// Изображение для хранения части графика.
    /// </summary>
    public Bitmap Image
    {
        get
        {
            return image;
        }
    }
}
```

Код данного класса показан в несколько сокращенном виде для концентрации внимания на ключевых моментах. Полный вариант данного кода содержится в прилагаемом к лабораторной работе проекте **ComplexVisual**. В частности, здесь не представлен исходный код вспомогательной функции для смешивания двух цветов. Данную функцию не требуется дорабатывать, но она необходима для реализации основных методов. Для более подробного изучения можно обратиться к исходному коду.

Каждое задание сопровождается комментарием, начинающимся со слова **TODO**. Все комментарии, начинающиеся с данного слова, среда Microsoft Visual Studio помечает специальным образом и заносит в список заданий. Для того чтобы просмотреть список заданий, нужно выполнить команду **View | Task List**. В результате на экране появится окно **Task List** (рис. 8), в котором будут перечислены задания. С помощью данного окна удобно осуществлять навигацию по программному коду, поскольку щелчок на каждом задании приводит к отображению соответствующей его части. Чтобы убрать выполненное задание из списка достаточно исключить из комментария слово **TODO**.

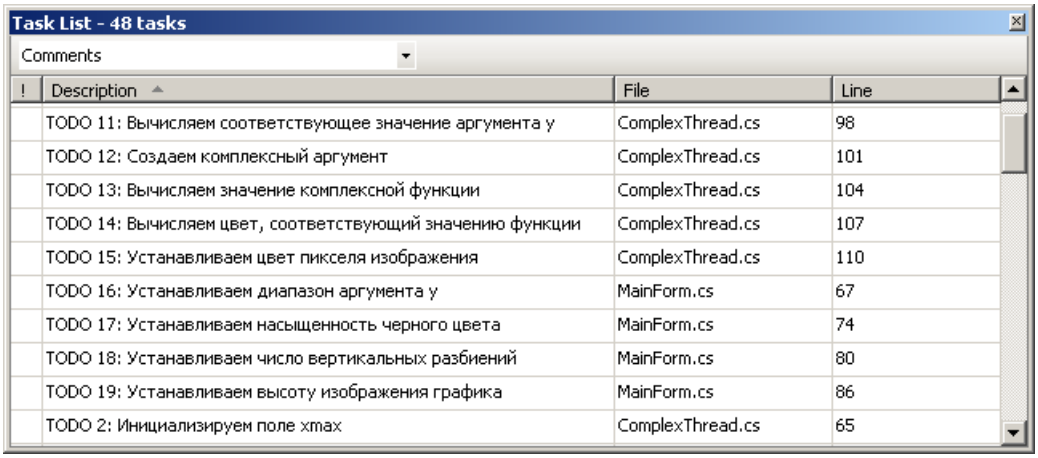


Рис. 8. Окно **Task List** позволяет осуществлять навигацию по назначенным заданиям

Приступим к доработке заготовленного кода до работоспособного состояния. Прежде всего, заметим, что перед объявлением класса необходимо подключить библиотеки **Alchemi** и **MathTools**. Далее следует объявление класса грид-потока как производного от абстрактного класса **`GThread`**, в котором определяется базовая функциональность для всех грид-потоков. Кроме того, перед объявлением класса необходимо поместить атрибут **`Serializable`** для того, чтобы объект был снабжен методами сериализации и десериализации.

При объявлении произвольного класса грид-потока необходимо определить все переменные, которые могут потребоваться *для вычислений* и последующего *восстановления результата* на машине пользователя из имеющихся фрагментов, вычисленных отдельными грид-потоками. В нашем случае для генерации части изображения необходимо задать его ширину и высоту, насыщенность белого и черного цвета, диапазоны аргументов *x* и *y*, определяющие область для визуализации, а также комплексную функцию. Результат выполненных вычислений сохраняется в точечный рисунок **`Bitmap`**. Соответствующий фрагмент кода представлен на приведенном ниже листинге.

```
/// <summary>Ширина обрабатываемой части изображения в пикселях.</summary>
private int width;

/// <summary>Высота обрабатываемой части изображения в пикселях.</summary>
private int height;

/// <summary>Минимальное значение аргумента x в обрабатываемой части области.</summary>
private double xmin;

/// <summary>Максимальное значение аргумента x в обрабатываемой части области.</summary>
private double xmax;

/// <summary>Минимальное значение аргумента y в обрабатываемой части области.</summary>
private double ymin;

/// <summary>Максимальное значение аргумента y в обрабатываемой части области.</summary>
private double ymax;

/// <summary>Комплексная функция для визуализации.</summary>
private BaseFunction function;

/// <summary>Насыщенность черного цвета в генерирумом изображении.</summary>
private double white;

/// <summary>Насыщенность белого цвета в генерирумом изображении.</summary>
private double black;

/// <summary>Точечный рисунок для хранения сгенерированной части изображения.</summary>
private Bitmap image;
```

Для восстановления результата на машине пользователя отдельные фрагменты изображения необходимо сложить в нужном порядке. Для этого необходимо сохранить горизонтальный и вертикальный номер обрабатываемой части. Соответствующие переменные представлены на приведенном ниже листинге.

```
/// <summary>Горизонтальный номер обрабатываемой части изображения.</summary>
private int hornumber;

/// <summary>Вертикальный номер обрабатываемой части изображения.</summary>
private int vernumber;
```

Далее необходимо создать *конструктор* для класса грид-потока, который будет вызываться для создания очередного экземпляра данного класса. Действия конструктора, как правило, сводятся к инициализации полей класса и выделению необходимых ресурсов. Ниже показана реализация конструктора для нашей задачи.

```
/// <summary>
/// Создает новый грид-поток для вычисления части графика.
/// </summary>
/// <param name="width">ширина части изображения</param>
/// <param name="height">высота части изображения</param>
/// <param name="hornumber">горизонтальный номер части изображения</param>
/// <param name="vernumber">вертикальный номер части изображения</param>
/// <param name="xmin">минимальное значение аргумента x в обрабатываемой области</param>
/// <param name="xmax">максимальное значение аргумента x в обрабатываемой области</param>
/// <param name="ymin">минимальное значение аргумента y в обрабатываемой области</param>
/// <param name="ymax">максимальное значение аргумента y в обрабатываемой области</param>
/// <param name="function">комплексная функция для визуализации</param>
/// <param name="white">насыщенность белого цвета</param>
/// <param name="black">насыщенность черного цвета</param>
public ComplexThread(int width, int height, int hornumber, int vernumber,
                     double xmin, double xmax, double ymin, double ymax,
                     BaseFunction function, double white, double black)
{
    // Устанавливаем ширину и высоту обрабатываемой части изображения
    this.width = width;
    this.height = height;

    // Устанавливаем горизонтальный и вертикальный номер обрабатываемой части изображения
    this.hornumber = hornumber;
    this.vernumber = vernumber;

    // Устанавливаем диапазоны аргументов в обрабатываемой части области
    this.xmin = xmin;
    this.xmax = xmax;
    this.ymin = ymin;
    this.ymax = ymax;

    // Устанавливаем функцию для визуализации
    this.function = function;

    // Устанавливаем насыщенность белого и черного цвета
    this.white = white;
    this.black = black;

    // Создаем точечный рисунок для хранения части графика
    image = new Bitmap(width, height);
}
```

Теперь все готово для реализации *основного* метода грид-потока, в котором будет производиться обработка заданной части общего изображения. Данный метод является обязательным для любого грид-потока, поэтому его объявление присутствует в абстрактном классе **`GThread`**, который является базовым для всех грид-потоков (комментарий к методу переведен на русский язык):

```
/// <summary>
/// Осуществляет исполнение грид-потока на удаленном Исполнителе. Данный
/// метод необходимо реализовывать в производных классах, добавляя
/// в него код, который должен выполняться на удаленных Исполнителях.
/// </summary>
public abstract void Start();
```

Таким образом, в каждой конкретной реализации класса грид-потока данный метод необходимо переопределять, заменяя ключевое слово **`abstract`** ключевым словом **`override`**.

В нашем учебном примере действия данного метода сводятся к последовательной обработке всех пикселей изображения, при этом для каждого пикселя необходимо вычислить соответствующую точку на комплексной плоскости и определить значение функции в этой точке. Затем для полученного значения функции следует вычислить соответствующий цвет и присвоить его обрабатываемому пикселю. Рассмотрим реализацию метода более подробно.

Прежде всего, вычислим шаги по двум аргументам x и y, соответствующие переходу к соседнему пикселю изображения:

```
// Вычисляем шаги по аргументам
double xstep = (xmax - xmin) / width;
double ystep = (ymax - ymin) / height;
```

Затем следует в двойном цикле перебрать все пиксели, вычислив для каждого пикселя соответствующее значение функции и закрасив его цветом, отвечающем данному значению:

```
// Последовательно обрабатываем все пиксели изображения
for (int j = 0; j < height; j++)
{
    // Вычисляем значение аргумента y
    double y = ymax - j * ystep;

    for (int i = 0; i < width; i++)
    {
        // Вычисляем значение аргумента x
        double x = xmin + i * xstep;

        // Создаем комплексный аргумент
        Complex arg = new Complex(x, y);

        // Вычисляем значение комплексной функции
        Complex value = function.CalcFunction(arg);

        // Вычисляем цвет, соответствующий значению функции
        Color color = CalcColor(value);

        // Устанавливаем цвет пикселя изображения
        image.SetPixel(i, j, color);
    }
}
```

Сразу оговоримся, что работа с классом точечного рисунка **`Bitmap`** не является достаточно эффективной. Для повышения производительности вычислений и сокращения времени передачи данных лучше воспользоваться целочисленным массивом. Однако используемый здесь подход отличается простотой и наглядностью, в то время как работа с целочисленным массивом потребовала бы дополнительного кода и усилий на его реализацию. Данное замечание следует учитывать при разработке реальных приложений.

Для того чтобы получить доступ к необходимым полям класса и при этом не нарушить инкапсуляцию данных, в конце класса определяются *свойства* для доступа к данным полям. Свойства являются именованными членами класса и предлагают гибкий механизм для чтения, записи и вычисления значений закрытых полей класса. На приведенном ниже листинге отдельно показаны все свойства, объявленные в классе грид-потока **`ComplexThread`**.

```
/// <summary>Вертикальный номер обрабатываемой части изображения.</summary>
public int VerNumber
{
    get
    {
        return vernumber;
    }
}

/// <summary>Горизонтальный номер обрабатываемой части изображения.</summary>
public int HorNumber
{
    get
    {
        return hornumber;
    }
}

/// <summary>Ширина обрабатываемой части изображения.</summary>
public int Width
{
    get
    {
        return width;
    }
}

/// <summary>Высота обрабатываемой части изображения.</summary>
public int Height
{
    get
    {
        return height;
    }
}

/// <summary>Изображение для хранения части графика.</summary>
```

```
public Bitmap Image
{
    get
    {
        return image;
    }
}
```

Прежде чем организовывать исполнение грид-потоков в вычислительной сети, построенной на базе инструментария Alchemi, желательно произвести тестирование вычислительного кода на локальной машине. Выявленные на этом этапе ошибки будет проще устранить. И лишь после того, как класс грид-потока будет полностью отлажен, можно организовывать взаимодействие с грид.

6.3. Задание 3 – Запуск грид-потоков на локальной машине

Формирование грид-потоков и организация их исполнения осуществляется в основном приложении **ComplexVisual**. В данном задании мы не будем рассматривать процесс проектирования пользовательского интерфейса, поскольку данный вопрос не имеет прямого отношения к теме лабораторной работы. Кроме того, предполагается, что читатель знаком с проектированием интерфейса с использованием библиотеки Windows Forms. Однако поскольку дальнейшие упражнения связаны с вводом исходных данных из различных элементов управления, рекомендуется ознакомиться с главной формой приложения (напомним, что проект прилагается к лабораторной работе).

Перейдем к рассмотрению программного кода главной формы. Для того чтобы открыть заготовку, предназначенную для дальнейшей доработки, выберите в окне **Solution Explorer** проект **ComplexVisual**, затем правой кнопкой мышки щелкните на файле **MainForm.cs** и в появившемся контекстном меню выберите команду **View Code**.

В первую очередь, необходимо подключить классы библиотек **Alchemi**, **ComplexThread** и **MathTools**. Далее в программе объявляются переменные, которые будут использоваться в процессе работы. Соответствующий фрагмент кода показан на приведенном ниже листинге.

```
using System;
using System.Drawing;
using System.Windows.Forms;
using Alchemi.Core;
using Alchemi.Core.Owner;
using GridThread;
using MathTools;

public partial class MainForm : Form
{
    ...

    /// <summary>Точечный рисунок для хранения графика комплексной функции.</summary>
    private Bitmap image;

    /// <summary>Время старта вычислений.</summary>
    private DateTime startTime;

    ...
}
```

На приведенном выше фрагменте кода показаны переменные, необходимые для запуска грид-потоков на *локальной* машине. Остальные переменные, необходимые для обработки грид-потоков в вычислительной сети, будут рассмотрены далее.

Нам предстоит доработать метод **`void BuildGraphLocal()`**, в котором осуществляется построение графика комплексной функции на локальном компьютере. Основные действия метода состоят в получении значений всех необходимых переменных, формировании нужного числа грид-потоков, запуске их на исполнение и получении результатов работы. Рассмотрим подробно реализацию данного метода.

Сначала необходимо установить минимальные и максимальные значения переменных x и y, которые определяют прямоугольную область, в которой будет производиться визуализация комплексной функции. Данные значения считываются из соответствующих элементов управления **`NumericUpDown`**, которые легко могут быть найдены на главной форме приложения.

```
// Устанавливаем диапазоны аргументов
double xmin = (double)spinnerXMin.Value;
double xmax = (double)spinnerXMax.Value;
double ymin = (double)spinnerYMin.Value;
double ymax = (double)spinnerYMax.Value;
```

Затем считываются значения насыщенности белого и черного цвета на генерируемом изображении. Чем выше значение насыщенности того или иного цвета, тем сильнее данный цвет будет выражен на изображении.

```
// Устанавливаем насыщенность белого и черного цвета
double white = (double)spinnerWhiteSaturation.Value;
double black = (double)spinnerBlackSaturation.Value;
```

Данная возможность оказывается очень полезной при анализе функций, значения которых либо слишком малы, либо слишком велики по абсолютной величине. В первом случае это приводит к тому, что картинка оказывается практически вся белая, во втором – практически вся черная. В результате и в том, и в другом случае анализ графика становится неудобным. Меняя значения насыщенности белого или черного цвета, можно добиться оптимального изображения.

Далее устанавливается число горизонтальных и вертикальных разбиений изображения на отдельные прямоугольные части.

```
// Устанавливаем число горизонтальных и вертикальных разбиений
int hor = (int)spinnerHorCells.Value;
int ver = (int)spinnerVerCells.Value;
```

Число горизонтальных и вертикальных разбиений определяет общее число генерируемых грид-потоков, каждый из которых создается для обработки одной части изображения. Например, если пользователь задал два разбиения по горизонтали и два разбиения по вертикали, то всего будет сгенерировано четыре грид-потока.

Затем следует установить ширину и высоту генерируемого изображения, а также выделить память, необходимую для его хранения. Созданное изображение отображается на графическом элементе управления **`PictureBox`**.

```
// Устанавливаем ширину и высоту изображения графика
int width = (int)spinnerWidth.Value;
int height = (int)spinnerHeight.Value;

// Создаем точечный рисунок для хранения изображения графика
image = new Bitmap(width, height);

// Отображаем точечный рисунок на элементе управления PictureBox
pictureBoxGraph.Image = image;
```

Наконец, необходимо сформировать функцию комплексного переменного на основе строки, введенной пользователем. Для этого нужно воспользоваться описанным выше статическим методом класса **`Parser`**. Не следует забывать, что введенная строка может оказаться некорректной, поэтому после формирования функции необходима проверка.

```
// Формируем комплексную функцию на основе введенной строки
BaseFunction funct = Parser.ParseString(textBoxFunction.Text);

// Проверяем, удалось ли сформировать функцию
if (funct == null)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("It is not possible to distinguish function. Check up input.");

    // Выходим из процедуры
    return;
}
```

Заключительный этап в реализации метода состоит в формировании и запуске грид-потоков. Однако сначала нужно вычислить значения некоторых переменных, необходимых для их создания. К ним относятся ширина и высота частей изображения в пикселях, а также ширина и высота соответствующих частей прямоугольной области, в которой строится функция, в координатах x и y. На приведенном ниже листинге показан соответствующий фрагмент кода.

```
// Вычисляем ширину и высоту частей изображения
int cellwidth = width / hor;
int cellheight = height / ver;

// Вычисляем изменения аргументов в частях области
double xstep = (xmax - xmin) / hor;
double ystep = (ymax - ymin) / ver;
```

Непосредственно перед запуском грид-потоков необходимо сохранить текущее время, чтобы впоследствии получить общее время вычислений, а также установить текущее и максимальное значение индикатора хода выполнения программы **`ProgressBar`**. Сделать это можно следующим образом:

```
// Обновляем текущее и максимальное значение полосы прогресса
progressBarBuildGraph.Value = 0;
progressBarBuildGraph.Maximum = hor * ver;

// Сохраняем время начала вычислений
startTime = DateTime.Now;
```

Далее следует организовать двойной цикл по всем горизонтальным и вертикальным разбиениям, в котором будут создаваться и запускаться грид-потоки. Соответствующий фрагмент кода показан на приведенном ниже листинге. Данный фрагмент кода завершает реализацию функции **`void BuildGraphLocal()`**.

```
// Формируем грид-потоки и исполняем их на локальной машине
for (int hornumber = 0; hornumber < hor; hornumber++)
{
    for (int vernumber = 0; vernumber < ver; vernumber++)
    {
        // Создаем грид-поток для обработки части графика
        ComplexThread thread = new ComplexThread(cellwidth, cellheight,
                                                 hornumber, vernumber,
                                                 xmin + xstep * hornumber,
                                                 xmin + xstep * (hornumber + 1),
                                                 ymax - ystep * (vernumber + 1),
                                                 ymax - ystep * vernumber,
                                                 funct, white, black);
        // Запускаем грид-поток на выполнение
        thread.Start();

        // Обновляем прогресс вычислений
        UpdateProgress(thread);
    }
}
```

После того, как грид-поток заканчивает обработку части изображения, следует обновить значение индикатора хода вычислений, время, затраченное на вычисления, и изображение на графическом элементе управления **`PictureBox`**. Все эти операции производятся во вспомогательном методе **`void UpdateProgress(GThread thread)`**. Остановимся подробнее на реализации данного метода.

В первую очередь обновляется значение индикатора хода вычислений и время, затраченное на вычисления. Заметим, что время вычислений выводится в заголовок главного окна приложения.

```
// Обновляем текущий прогресс
progressBarBuildGraph.Value++;

// Обновляем затраченное время
Text = "Complex Visual - " + (DateTime.Now - startTime).ToString();
```

Затем обрабатываются результаты вычислений успешно завершенного грид-потока, который передается в качестве параметра функции **`void UpdateProgress(GThread thread)`**. Часть графика, рассчитанную данным грид-потоком, необходимо добавить к общему изображению. Легко видеть, что параметр функции имеет абстрактный тип **`GThread`**, поэтому, прежде чем использовать успешно завершенный грид-поток, его необходимо привести к типу **`ComplexThread`**.

```
// Получаем упешно завершенный грид-поток
ComplexThread complexThread = (ComplexThread)thread;
```

Далее вычисляются координаты левого верхнего пикселя части изображения, и производится его копирование на общее изображение графика функции. Для работы с изображением используется стандартный класс **`Graphics`**.

```
// Вычисляем координаты левого верхнего угла части графика
int startX = complexThread.HorNumber * complexThread.Width;
int startY = complexThread.VerNumber * complexThread.Height;

// Создаем объект Graphics для работы с изображением
Graphics graphics = Graphics.FromImage(image);

// Добавляем к общему изображению сгенерированную часть
graphics.DrawImage(complexThread.Image, startX, startY);
```

Чтобы на экране компьютера пользователь смог увидеть изменения, содержимое элемента управления **`PictureBox`** следует принудительно обновить. Сделать это можно следующим образом:

```
// Обновляем изображение комплексной функции
pictureBoxGraph.Refresh();
```

Обработка грид-потоков на локальной машине завершена. В заключение заметим, что использование нескольких грид-потоков в данном случае не приводит к какому-либо ускорению вычислений. Это связано с тем, что все грид-потоки выполняют свою работу последовательно, а не одновременно. Следующий грид-поток запускается лишь тогда, когда предыдущий завершает свою работу. Однако такой подход позволит нам в дальнейшем добавить обработку грид-потоков в вычислительной грид с минимальными изменениями кода. Кроме того, однопоточное приложение существенно легче отлаживать по сравнению с многопоточным.

6.4. Задание 4 – Тестирование приложения на локальной машине

Теперь приложение можно запустить и проверить работоспособность вычислительного кода грид-потоков. Для этого выполните команду **Debug | Start Debugging** или нажмите клавишу **F5**. Если в процессе выполнения предыдущих заданий не было допущено синтаксических ошибок, программа успешно откомпилируется и запустится. На рис. 9 показано главное окно приложения.

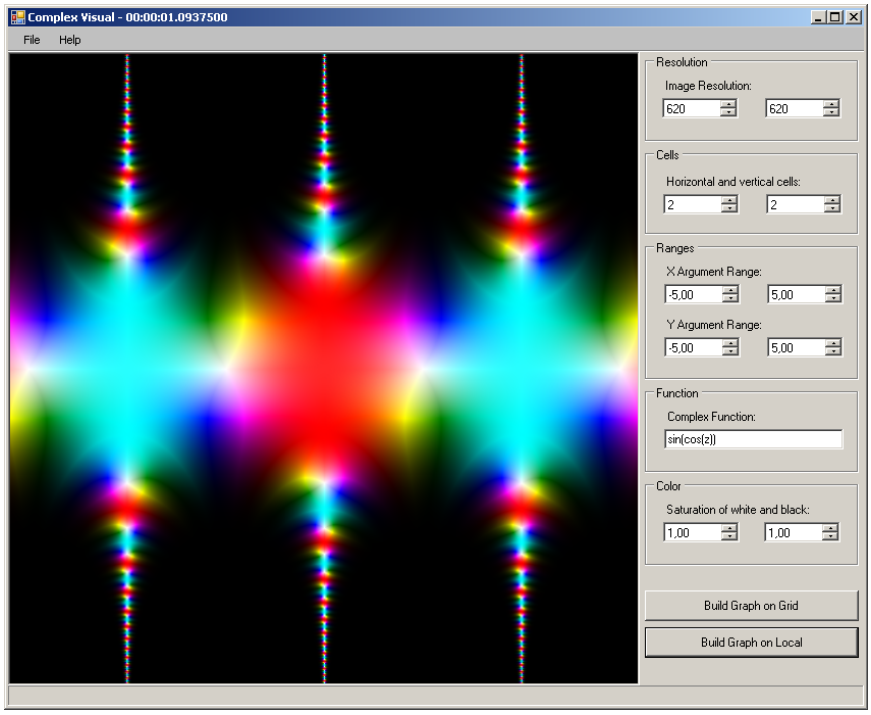


Рис. 9. Главное окно приложения после первого запуска

В правой части главного окна располагаются элементы управления для ввода всех параметров, необходимых для работы программы. Отдельные параметры объединены в логические группы. В группе **Resolution** содержатся поля ввода для установки размеров генерируемого изображения. Следует иметь в виду, что с увеличением размеров изображения увеличивается и время работы программы. В следующей группе (**Cells**) представлены поля ввода для настройки числа горизонтальных и вертикальных разбиений изображения, влияющих на общее число грид-потоков. Далее идет группа параметров **Ranges**, в которой содержатся поля ввода для установки границ прямоугольной области, в которой будет визуализироваться комплексная функция. Аналитическое выражение самой комплексной функции можно задать в группе **Function**, содержащей единственное поле ввода. Наконец, в группе **Color** представлены поля ввода для установки дополнительных параметров, задающих насыщенность белого и черного цвета в генерируемом изображении. В большинстве случаев данные параметры не требуется менять, однако, если значения функции в указанной области слишком малы или, наоборот, слишком велики (по абсолютной величине), то настройка данных параметров может значительно повысить наглядность генерируемого изображения. Заметим, что все параметры программы имеют значения по умолчанию, которые отображаются в соответствующих элементах управления при первом запуске.

Для того чтобы произвести тестирование грид-потоков на локальной машине, щелкните на кнопке **Build Graph on Local**. В результате на кране появится изображение сгенерированной функции. Можно поэкспериментировать с входными данными, изменив комплексную функцию, область визуализации, параметры насыщенности белого и черного цвета. В результате можно получить очень интересные изображения.

Однако наша цель состоит в том, чтобы запускать грид-потоки в вычислительной сети, построенной на базе инструментария Alchemi. В следующем задании мы соответствующим образом доработаем код.

6.5. Задание 5 – Запуск грид-потоков в вычислительной грид

Для организации исполнения грид-потоков в вычислительной грид необходимо объявить некоторые новые переменные. В первую очередь нужно объявить экземпляр грид-приложения, посредством которого осуществляется взаимодействие с грид.

```
/// <summary>Экземпляр грид-приложения для взаимодействие с грид.</summary>
private GApplication gridApplication;
```

При первом запуске вычислений необходимо установить подключение к грид. Данная процедура должна выполняться один раз. Для того чтобы избежать повторных подключений, объявим соответствующую логическую переменную:

```
/// <summary>Отвечает за инициализацию грид-приложения.</summary>
private bool gridInit = false;
```

Все необходимые сервисные операции инструментарий Alchemi выполняет в *отдельном* рабочем потоке, периодически посылая уведомления в *основной* поток приложения. Уведомления могут посылаться при успешном или неудачном завершении грид-потока, а также при завершении работы грид-приложения. Трудность состоит в том, что библиотека пользовательского интерфейса Windows Forms работает с окном и его элементами управления в специально выделенном *основном* потоке. Иными словами, из “внешнего” потока невозможно получить доступ к окну и его элементам управления. Тем не менее, для нормальной работы программы это необходимо. В рассматриваемом примере программа должная получать уведомления (обрабатывать события) об успешном завершении некоторого грид-потока и выводить в главное окно обновленную картинку и значение индикатора хода выполнения. Таким образом, для реализации взаимодействия необходимо использовать средства синхронизации, предоставляемые данной библиотекой. В библиотеке Windows Forms одним из таких средств является метод базового класса **`Control`**:

```
public virtual IAsyncResult BeginInvoke(Delegate, object[]);
```

Данный метод вызывает заданный делегат с определенным набором параметров из *основного* потока приложения. Как видно из объявления метода, для нормального использования механизма синхронизации необходимо для каждого уведомления определить свой тип делегата и использовать для его вызова указанный выше метод, что является определенным неудобством библиотеки Windows Forms. Далее мы проиллюстрируем все вышеизложенное соответствующими фрагментами кода.

Как уже было сказано, в нашем случае требуется обрабатывать уведомление об успешном завершении грид-потока. В предыдущей реализации для этой цели служил метод **`void UpdateProgress(GThread thread)`**. В новой реализации данный метод должен вызываться из “внешнего” потока, поэтому для него следует создать специальный делегат и объявить экземпляр данного делегата. Соответствующий фрагмент кода показан на приведенном ниже листинге.

```
/// <summary>
/// Делегат для обновления прогресса вычислений.
/// </summary>
/// <param name="thread">успешно завершившийся грид-поток</param>
private delegate void UpdateProgressDelegate(GThread thread);

/// <summary>Экземпляр делегата для обновления прогресса вычислений.</summary>
private UpdateProgressDelegate updateProgressDelegate;
```

Экземпляр делегата для обновления прогресса вычислений инициализируется при создании главного окна приложения:

```
/// <summary>
/// Создает основную форму приложения.
/// </summary>
public MainForm()
{
    InitializeComponent();

    // Создаем делегат для обновления прогресса вычислений
    updateProgressDelegate = new UpdateProgressDelegate(UpdateProgress);
}
```

В этом задании нам предстоит доработать метод **`void BuildGraphGrid()`**, в котором организуется обработка грид-потоков в вычислительной сети, построенной на базе инструментария Alchemi. Заметим, что реализация данного метода очень близка к реализации соответствующего метода для запуска грид-

потоков на локальной машине. Различие состоит в дополнительном коде для установки соединения с грид и для работы с классом грид-приложения. Остановимся лишь на этих фрагментах кода, так как остальная его часть была рассмотрена ранее.

Прежде чем запускать грид-потоки на выполнение в вычислительной грид, необходимо установить подключение к одному из Менеджеров и произвести инициализацию грид-приложения. Напомним, что Менеджер представляет собой контроллер сегмента грид и координирует работу нескольких Исполнителей. Соединение с грид достаточно устанавливать лишь один раз для каждого сеанса работы с программой. Сделать это можно, например, при первом запуске вычислений. Чтобы избежать многократных подключений, проверяется значение соответствующей логической переменной. На приведенном ниже листинге показан код установки соединения и инициализации грид-приложения.

```
// Проверяем, требуется ли инициализация грид-приложения
if (!gridInit)
{
    // Создаем диалог для ввода параметров соединения
    GConnectionDialog connectionDialog = new GConnectionDialog();

    // Отображаем диалог
    if (connectionDialog.ShowDialog() == DialogResult.OK)
    {
        // Создаем новое грид-приложение многократного использования
        gridApplication = new GApplication(true);

        // Устанавливаем имя созданного грид-приложения
        gridApplication.ApplicationName = "Complex Visual - Alchemi sample";

        // Устанавлливаем соединение созданного грид-приложения
        gridApplication.Connection = connectionDialog.Connection;

        // Устанавливаем зависимости, необходимые для работы грид-потоков
        gridApplication.Manifest.Add(new ModuleDependency(typeof(MathTools.Complex).Module));
        gridApplication.Manifest.Add(new
                            ModuleDependency(typeof(GridThread.ComplexThread).Module));

        // Добавляем событие для обработки успешно завершившихся грид-потоков
        gridApplication.ThreadFinish += new GThreadFinish(ThreadFinish);

        // Добавляем событие для обработки неудачно завершившихся грид-потоков
        gridApplication.ThreadFailed += new GThreadFailed(ThreadFailed);

        // Инициализация завершена
        gridInit = true;
    }
}
```

Дадим некоторые пояснения к данному фрагменту кода.

Для установки подключения к грид разработчики инструментария Alchemi предусмотрели специальное диалоговое окно. В диалоге **Alchemi Grid Connection** (рис. 10) пользователю предлагается ввести основные параметры подключения, такие как:

- адрес и порт Менеджера, к которому производится подключение;
- имя пользователя и пароль.

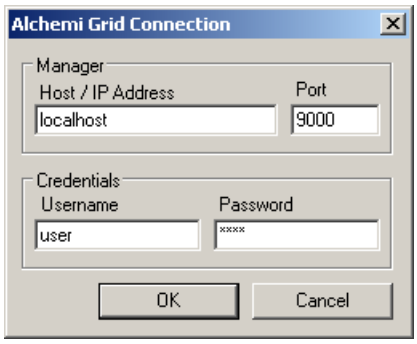


Рис. 10. Диалоговое окно подключения к грид

После получения основных параметров подключения производится инициализация грид-приложения, то есть создается новый экземпляр грид-приложения, устанавливается его имя и

подключение, добавляются модули, необходимые для запуска грид-потоков на удаленных Исполнителях, а также обработчики некоторых событий.

Инструментарий Alchemi предусматривает несколько событий, которые можно обрабатывать. Ключевыми являются события **`ThreadFinish`** и **`ThreadFailed`**, возникающие соответственно при успешном и неудачном завершении грид-потока. Важно отметить, что следует *всегда* обрабатывать событие **`ThreadFinish`**, которое позволяет получить результаты работы грид-потока на удаленном Исполнителе. Поскольку вычислительная сеть может оказаться ненадежной, некоторые грид-потоки могут неудачно заканчивать свою работу. В этом случае рекомендуется реализовать также обработчик события **`ThreadFailed`**, который может осуществлять, например, перезапуск неудачно завершенного грид-потока на доступных ресурсах.

После инициализации грид-приложения следует установить значения всех переменных, которые потребуются для формирования грид-потоков. Этот код полностью идентичен соответствующему коду для запуска грид-потоков на локальной машине и может быть взят из предыдущего упражнения.

Далее обычным образом формируются грид-потоки. Данная часть кода остается такой же, как и в предыдущем упражнении, поэтому мы не будем подробно на этом останавливаться. Отличие состоит в том, что грид-потоки не запускаются сразу же после их создания, а добавляются к экземпляру грид-приложения.

```
// Формируем грид-потоки для обрабоки отдельных частей графика
for (int hornumber = 0; hornumber < hor; hornumber++)
{
    for (int vernumber = 0; vernumber < ver; vernumber++)
    {
        // Создаем новый грид-поток
        ComplexThread thread =
                          new ComplexThread(cellwidth, cellheight,
                                            hornumber, vernumber,
                                            xmin + xstep * hornumber,
                                            xmin + xstep * (hornumber + 1),
                                            ymax - ystep * (vernumber + 1),
                                            ymax - ystep * vernumber,
                                            funct, white, black);

        // Добавляем созданный грид-поток к грид-приложению
        gridApplication.Threads.Add(thread);
    }
}
```

Заметим, что добавление грид-потока к грид-приложению не приводит к его запуску. Более того, на этом этапе все грид-потоки хранятся на локальном компьютере и не пересылаются на удаленные Исполнители.

Заключительный этап разработки метода состоит в запуске грид-приложения. Однако перед этим следует проверить, работает ли оно в данный момент. Если это так, то грид-приложение следует корректно остановить. Поскольку данное действие может генерировать ряд исключений, его следует заключить в блок **`try-catch`**. Соответствующий фрагмент кода показан ниже.

```
// Проверяем, работает ли грид-приложение
if (gridApplication.Running)
{
    try
    {
        // Останавливаем грид-приложение
        gridApplication.Stop();
    }
    catch (Exception e)
    {
        // Выводим сообщение об ошибке
        MessageBox.Show("Error trying to stop already running grid application: " +
                                                                        e.ToString());
        // Выходим из процедуры
        return;
    }
}
```

Непосредственно перед запуском грид-приложения необходимо сохранить текущее время и установить текущее и максимальное значение индикатора хода вычислений, равное общему числу сгенерированных грид-потоков. Данный код полностью повторяет соответствующий код из предыдущего упражнения.

Наконец, необходимо запустить грид-приложение на выполнение. При этом грид-потоки будут отправлены по сети к указанному Менеджеру, который будет координировать их исполнение в грид,

построенной на базе Alchemi. Следует учитывать, что на этом этапе возможны различные типы ошибок и исключений, поэтому необходимо обеспечить механизм их обработки. Соответствующий фрагмент кода показан ниже.

```
try
{
    // Запускаем грид-приложение
    gridApplication.Start();
}
catch (Exception e)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Error trying to run grid application: " + e.ToString());
}
```

При успешном выполнении всех указанных действий грид-потоки будут отправлены по сети на удаленные Исполнители для обработки. При этом при успешном завершении грид-потока будет вызываться обработчик события **`ThreadFinish`**, а при неудачном – обработчик события **`ThreadFailed`**. Рассмотрим данные обработчики подробнее.

Действия обработчика события **`ThreadFinish`** сводятся к вызову метода **`void UpdateProgress(GThread thread)`** из *основного* потока приложения. Возможная реализация представлена на приведенном ниже листинге.

```
/// <summary>
/// Обработчик события, возникающего при успешном завершении грид-потока.
/// </summary>
/// <param name="thread">успешно завершенный грид-поток</param>
void ThreadFinish(GThread thread)
{
    // Вызываем делегат для обновления прогресса обработки изображения
    BeginInvoke(updateProgressDelegate, new object[] { thread });
}
```

В обработчике события **`ThreadFailed`** ограничимся выводом сообщения о неудачном завершении грид-потока и не будем предпринимать каких-либо действий для исправления ситуации. Поскольку в таком обработчике не требуется получать доступ к элементам управления главного окна, то создавать новый делегат необязательно. Соответствующий фрагмент кода приведен ниже.

```
/// <summary>
/// Обработчик события, возникающего при неудачном завершении грид-потока.
/// </summary>
/// <param name="thread">неудачно завершившийся грид-поток</param>
void ThreadFalied(GThread thread, Exception e)
{
    MessageBox.Show("Thread with Id = " + thread.Id + " failed: " + e.ToString());
}
```

Это простейшее поведение программы, однако правильнее было бы проанализировать случившуюся ошибку (для этого служит объект **`Exception`**) и повторно перезапустить неудачно завершившийся грид-поток на доступных Исполнителях. Возможность улучшить обработку неудачно завершившихся грид-потоков предоставляется выполнить самостоятельно.

На этом доработку метода можно считать завершенной. Заметим, что при выходе из программы объект грид-приложения желательно удалить. В противном случае на Менеджере останутся связанные с ним ресурсы, что оказывается весьма нежелательным, поскольку данная компонента системы продолжает функционировать и после завершения пользовательской программы. Данный код можно поместить, например, в обработчик закрытия главного окна приложения:

```
// Уничтожаем грид-приложение
if (gridApplication != null)
{
    gridApplication.Dispose();
}
```

В следующем задании мы протестируем работу грид-потоков в вычислительной грид, построенной на базе инструментария Alchemi.

6.6. Задание 6 – Тестирование приложения в вычислительной грид

Теперь приложение можно запустить и протестировать его работу в вычислительной грид, построенной на базе инструментария Alchemi. Для этого выполните команду **Debug | Start Debugging** или нажмите клавишу **F5**. Если в процессе выполнения предыдущих заданий не было допущено

синтаксических ошибок, программа успешно откомпилируется и запустится. На рис. 8 показано главное окно приложения.

Перед тем, как работать с программой, на машине разработчика следует создать минимальную “грид” из одного Менеджера и одного Исполнителя. Следует заметить, что при наличии многоядерной или многопроцессорной машины желательно запускать несколько Исполнителей для более оптимального использования вычислительных ресурсов. Если имеется возможность, отдельные Исполнители лучше запустить на различных компьютерах сети. Такое тестирование наиболее приближено к реальным условиям работы программы.

Для того чтобы произвести тестирование грид-потоков в вычислительной сети, построенной на базе инструментария Alchemi, щелкните на кнопке **Build Graph on Grid**. В результате на кране появится изображение сгенерированной функции. Можно поэкспериментировать с входными данными, изменив комплексную функцию, область визуализации, параметры насыщенности белого и черного цвета. В результате можно получить очень интересные изображения.

7. Упражнение 5 – Получение сведений о выполняемом распределенном приложении

Инструментарий Alchemi включает средства мониторинга ключевых параметров быстродействия вычислительной грид и запущенных приложений. Для этой цели служит приложение Alchemi Console, которое входит в стандартную поставку Alchemi SDK. В данном упражнении мы применим это приложение для контроля основных параметров выполнения разработанной программы Complex Visual. На рис. 11 показано главное окно приложения Alchemi Console.

Прежде всего, необходимо установить подключение к вычислительной грид. Для этого следует нажать на кнопку установки подключения на панели инструментов. Заметим, что можно воспользоваться командой **File | New Grid Connection**, однако на момент подготовки документа она была неработоспособна. Вероятно, в последующих версиях эта недоработка будет устранена. В результате проделанных действий на экране появится стандартное окно подключения к грид (рис. 10), в котором следует указать адрес и порт Менеджера, контролирующего работу сегмента грид, а также имя пользователя и пароль. Если все введенные параметры являются правильными и подключение пройдет успешно, то главное окно приложения Alchemi Console приобретет вид, показанный на рис. 12.

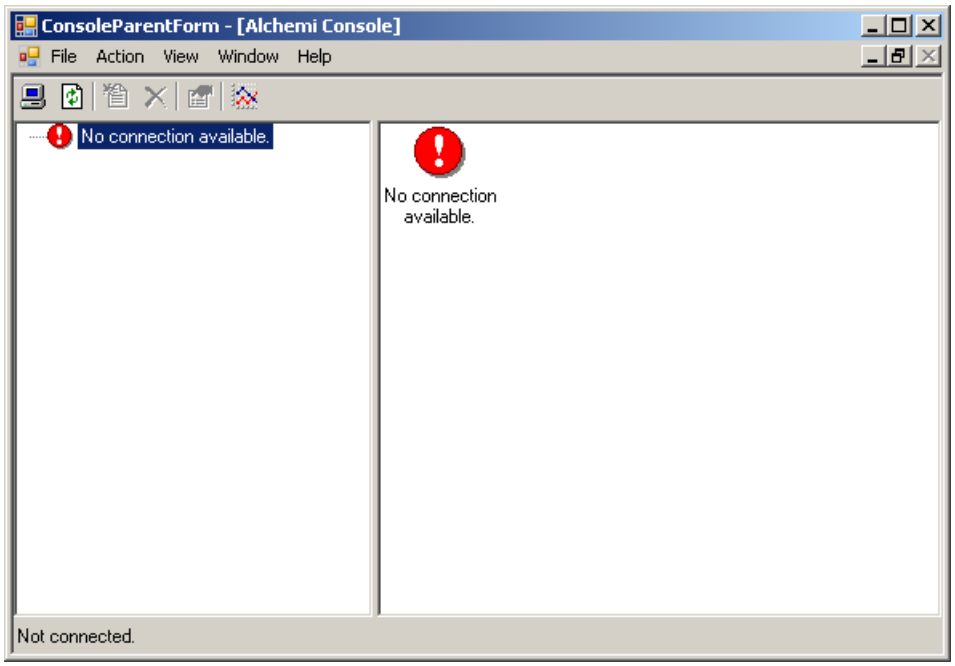


Рис. 11. Главное окно приложения Alchemi Console после запуска

Все действия, производимые с помощью приложения Alchemi Console, поделены на категории и представлены в виде иерархической структуры в левой части окна. Мы не будем подробно останавливаться на описании всевозможных действий, так как реализация большинства из них находится на начальном уровне, часто возникают ошибки при работе или, наоборот, ничего не происходит. Однако средства мониторинга уже реализованы и вполне стабильно функционируют.

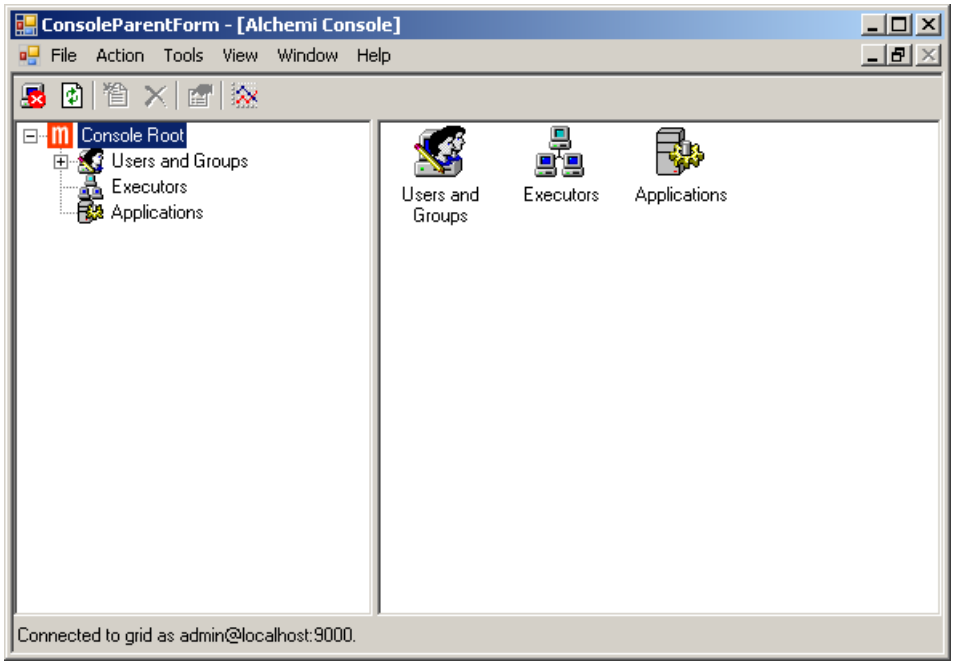


Рис. 12. Главное окно приложения Alchemi Console после установки подключения

Для просмотра списка доступных Исполнителей выберите категорию **Executors** в левой части окна. В результате в правой части окна отобразятся все доступные в данный момент Исполнители (рис. 13).

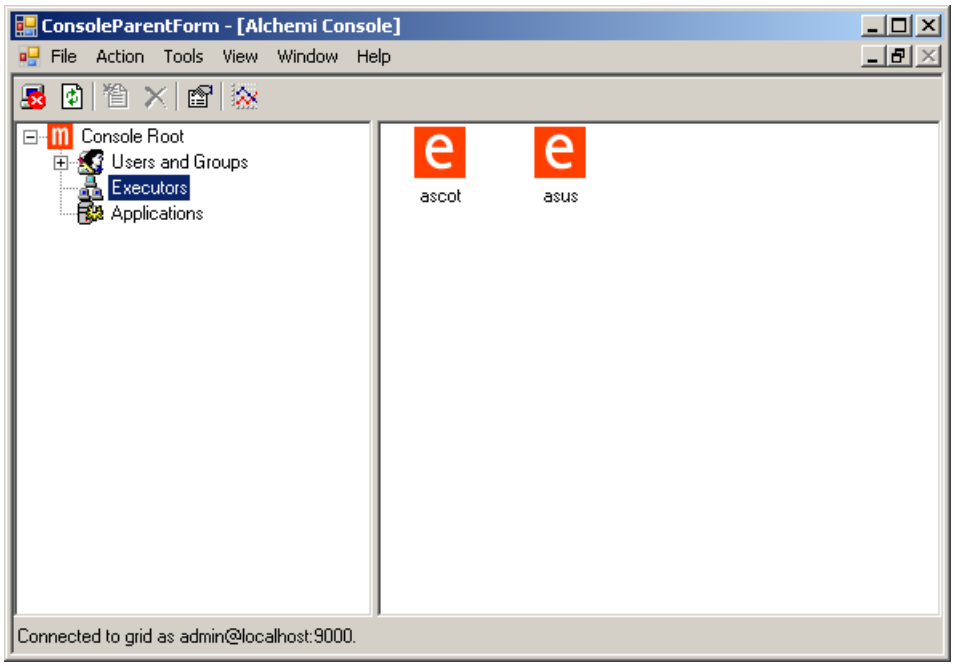


Рис. 13. Информация о доступных в данный момент Исполнителях

Для получения более подробных сведений о каждом Исполнителе щелкните на соответствующем значке. На экране появится окно с подробной информацией о выбранном Исполнителе (рис. 14), в частности, информация о версии операционной системы, центральном процессоре, быстродействии.

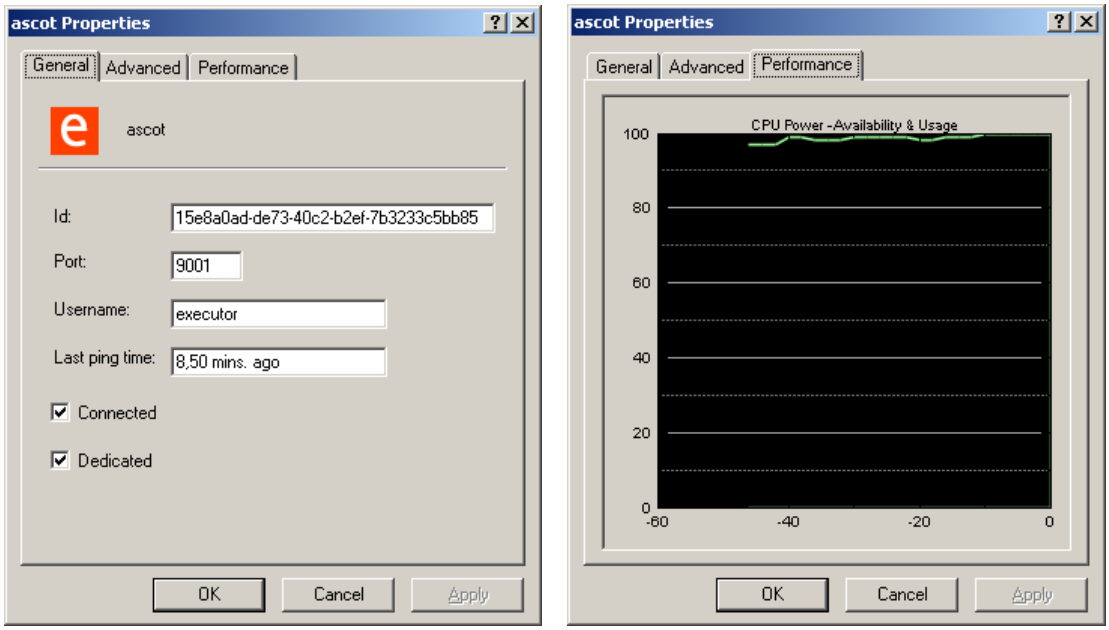


Рис. 14. Окно с подробными сведениями о выбранном Исполнителе

Запустите на выполнение разработанное приложение Complex Visual и выберите категорию **Applications** в левой части окна Alchemi Console. В списке приложений появится элемент **Complex Visual – Alchemi Sample**. Напомним, что именно такое имя было присвоено приложению при инициализации:

```
// Устанавливаем имя созданного грид-приложения
gridApplication.ApplicationName = "Complex Visual - Alchemi sample";
```

Если выбрать в списке данное приложение, то в правой части окна Alchemi Console появится список всех грид-потоков, запущенных на выполнение в грид-сети (рис. 15).

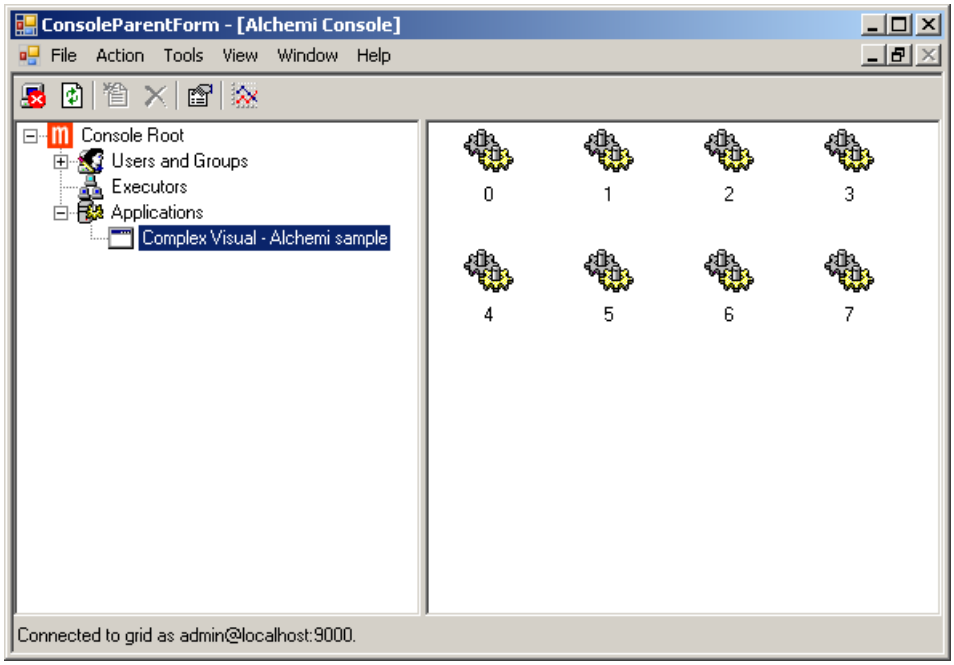


Рис. 15. Информация о запущенных приложениях и соответствующих грид-потоках

Для просмотра более подробной информации о каждом грид-потоке, щелкните на нем дважды левой кнопкой мышки. В результате откроется окно свойств выбранного грид потока (рис. 16). Для каждого грид-потока отображается идентификатор соответствующего грид-приложения, время создания и завершения, Исполнитель, на котором производились вычисления, текущее состояние грид-потока и его приоритет. Кроме того, для запущенных грид-потоков имеется возможность принудительного завершения с помощью кнопки **Abort**.

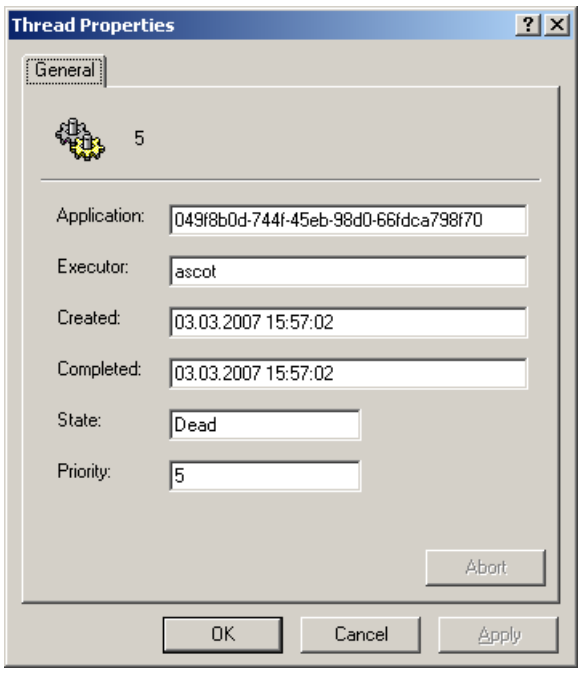


Рис. 16. Окно свойств выбранного грид-потока

В тех случаях, когда требуется получить информацию о быстродействии вычислительной грид в целом, можно воспользоваться окном **System Performance Monitor**, на котором отображается информация о суммарной вычислительной мощности, количестве доступных Исполнителей, запущенных грид-приложениях и грид-потоках (рис. 17). Для вызова данного окна можно воспользоваться кнопкой на панели инструментов (крайняя справа).

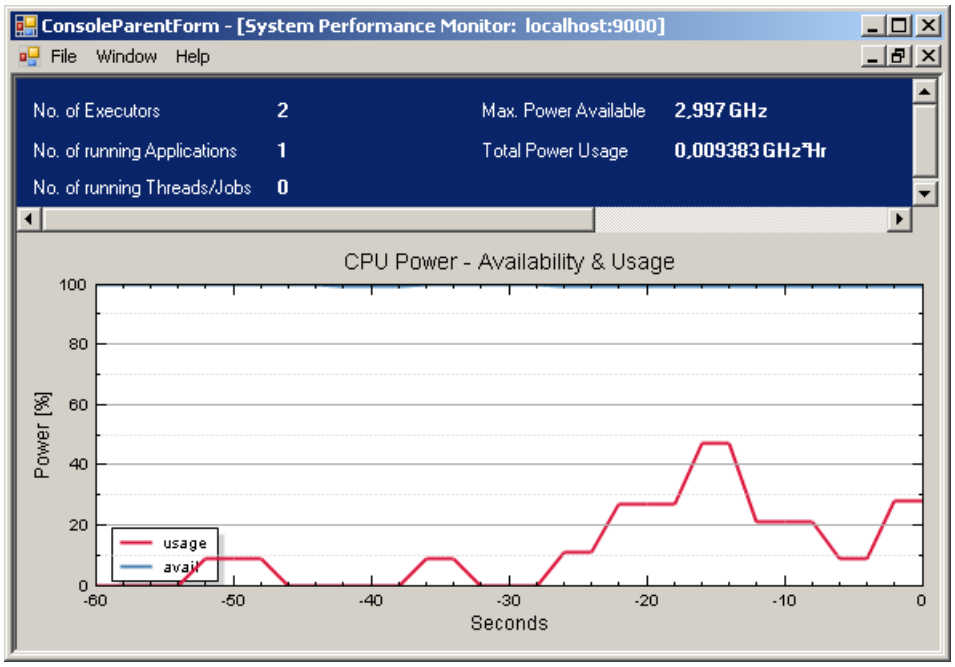


Рис. 17. Окно с общими показателями быстродействия грид-системы

В заключение еще раз заметим, что приложение Alchemi Console на момент написания данного документа находилось на ранней стадии разработки. Этот факт следует учитывать при работе и с пониманием относится к возможным ошибкам и недоработкам. Тем не менее, основная информация о грид-системе в целом и об отдельных грид-приложениях уже сейчас предоставляется вполне стабильно.

8. Заключение

В данной лабораторной работе рассматривался процесс разработки нового приложения для грид, построенной на базе инструментария Alchemi. В качестве учебной задачи была выбрана задача визуализации функций комплексного переменного.

Очевидно, что инструментарий Alchemi предлагает довольно обширные и простые в реализации возможности для разработки грид-приложений. К преимуществам инструментария Alchemi можно отнести:

- *Простота реализации и компактность кода.*

  Обеспечивается благодаря развитой архитектуре высокоуровневых классов Microsoft .NET Framework. У разработчика имеются в распоряжении высокоуровневые инструменты для выполнения практически любых сервисных операций, связанных с чтением и записью файлов, запуском приложений и контролем их исполнения и т.д.

- *Возможность быстро и удобно проектировать пользовательский интерфейс.*

  Инструментарий Alchemi использует все возможности библиотеки пользовательского интерфейса Windows Forms.

- *Легкость в отладке и использовании приложений.*

  Разработанные классы грид-потоков можно сначала отладить на локальной машине, и лишь затем осуществлять их запуск в грид.

- *Возможность запуска разработанных приложений на локальных компьютерах.*

  Разработанные приложения можно запускать на локальных компьютерах без установленных компонент инструментария Alchemi. Любая программа для инструментария Alchemi может использоваться как обычное приложение для операционной системы Microsoft Windows с установленной платформой Microsoft .NET Framework (при этом, конечно, приложение не получит доступ к ресурсам грид).

Однако при использовании инструментария Alchemi можно обнаружить некоторые недостатки. Отметим на наш взгляд наиболее критичные из них:

- *Отсутствие переносимости кода и ориентация на единственную платформу.*

  К сожалению, на сегодняшний день отсутствует полноценная замена платформы Microsoft .NET Framework для других операционных систем. Данный факт делает невозможным запуск программ, написанных с использованием инструментария Alchemi, под другими операционными системами. Стоит заметить, что прогресс в этом направлении не стоит на месте, и межплатформенная разработка Mono[1] уже сейчас предлагает неплохой уровень совместимости с платформой Microsoft .NET Framework.

- *Ограниченный класс задач из-за отсутствия обмена данными между грид-потоками.*

  На сегодняшний день инструментарий Alchemi не имеет поддержки средств обмена данными между грид-потоками. Данный факт несколько ограничивает круг применения инструментария. Согласно планам разработчиков, поддержка данного механизма должна появиться в будущем. Следует, однако, заметить, что отсутствие подобных средств нельзя зачислить в серьезные недостатки инструментария, так как для задач, решаемых в грид, требование независимости подзадач является вполне естественным. Это позволяет существенно упростить контроль над выполнением приложения и гарантировать успешное продолжение вычислений даже в случае отказа отдельных Исполнителей. Кроме того, простейшие способы обмена данными предложить все же можно[2].

- *Слабая масштабируемость инструментария.*

  Инструментарий Alchemi лучше всего подходит для объединения в вычислительную сеть нескольких десятков компьютеров. При подключении большего числа компьютеров могут появиться проблемы со стабильностью системы (в частности, со стабильностью работы Менеджера). Улучшить масштабируемость позволяет организация иерархической структуры из отдельных Менеджеров. Опять же, это скорее особенность, которую следует учитывать, а не недостаток инструментария Alchemi.

---

[1] Главной задачей проекта Mono (http://www.mono-project.com) является создание полноценной реализации среды разработки Microsoft .NET Framework для UNIX-подобных операционных систем. Инициативу Mono возглавляет Мигель де Иказа, один из участников проекта Gnome. Спонсирует же группу разработчиков Mono корпорация Novell.

[2] Например, обмен данными между грид-потоками можно произвести в обработчике события `ThreadFinish`, после чего запустить грид-потоки вновь.

9. Вопросы

1. Какие модели программирования предоставляет инструментарий Alchemi? В чем их особенности?
2. В каких ситуациях следует применять модель грид-потоков?
3. В чем состоят основные этапы разработки приложения с помощью модели грид-потоков?
4. Каким требованиям должна удовлетворять вычислительная схема?
5. Опишите два стандартных вида параллелизма.
6. Опишите два стандартных вида вычислительных схем.
7. В чем состоят основные этапы разработки класса грид-потока?
8. Каким образом лучше всего отлаживать класс грид-потока?
9. Каким образом осуществляется запуск грид-потоков в вычислительной сети?
10. Какие средства предоставляет инструментарий Alchemi для мониторинга грид-приложений?

10. Упражнения

Разработанную программу можно значительно усовершенствовать, добавив в нее ряд новых возможностей.

1. Добавить возможность остановки вычислений при их запуске как на локальном компьютере, так и в вычислительной грид.
2. Реализовать корректную обработку неудачно завершившихся грид-потоков и последующий их перезапуск на доступных Исполнителях (если это возможно).
3. В качестве другого дополнительного упражнения можно предложить модифицировать программу таким образом, чтобы она строила различные фрактальные множества, такие как множество Мандельброта и Джулии (подобная модификация не займет много времени). Подробные сведения о фрактальных множествах можно найти в работе [2].

11. Литература

1. Ахо, Альфред, В., Сети, Рави, Ульман, Джеффри, Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2003. – 768 с.
2. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики. – СПб.: БХВ-Петербург, 2003. – 560 с.
3. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. http://www.gridbus.org/~alchemi/files/alchemi_bookchapter.pdf
4. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-based Enterprise Grid Computing System. http://www.gridbus.org/%7Eraj/papers/alchemi_icomp05.pdf
5. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-based Grid Computing Framework and its Integration into Global Grids. http://www.gridbus.org/~alchemi/files/alchemi_techreport.pdf
6. http://www.alchemi.net – официальный сайт проекта Alchemi