Лабораторная работа 5

Изучение методики визуализации данных

Цель работы:

Изучить методики визуализации данных. Изучить наиболее простой и в то же время универсальный способ визуализации данных с помощью библиотеки G2. Закрепить навыки программирования с использованием MPI.

Задачи работы:

1) Изучить теоретические основы методики визуализации данных.
2) Выполнить задание лабораторной работы.
3) Ответить на вопросы.

Теория

Одним из важных моментов при проведении численных экспериментов является визуализация полученных данных. Когда промежуточных результатов мало, например, важно отобразить лишь факт окончания очередной итерации, можно ограничиться консольным (текстовым) выводом. В этом случае никаких особенных инструментов не требуется. Вывод данных осуществляется стандартными средствами языка. Так же в качестве отдельной задачи можно рассматривать вопрос обработки конечных результатов счета, сохраненных в файле. Для операционной системы Linux существует достаточно широкий спектр приложений, которые можно использовать для визуальной обработки данных, полученных в результате выполнения расчетной задачи. В качестве таких приложений можно упомянуть как различные электронные таблицы типа *OpenOffice.org Calc*, *Kspread* и др., так и широкий список специализированных пакетов, например *Grapher*.

В лабораторной работе мы рассмотрим методики оперативного визуального отображения данных непосредственно в процессе счета задачи.

Зачастую возникает необходимость визуализировать промежуточные результаты, полученные по окончании каждой отдельной итерации. Например при проведении численных экспериментов, связанных с решением задач электо-, газо- или термодинамики, может появиться желание посмотреть эволюцию исследуемого процесса во времени. Иногда полученная картинка может повлиять на решение о целесообразности дальнейшего счета. К примеру, если увиденная динамика процесса далека от ожидаемой, то это может свидетельствовать о возможных ошибках в численном алгоритме или неверных начальных и граничных условиях задачи.

Поскольку визуализация должна осуществляться непосредственно во время счета, использование отдельных специализированных пакетов графической обработки данных отпадает. Средства визуализации должны быть встроены в программу как часть численного алгоритма. В Linux существует огромное количество различных библиотек работы с графикой как в режиме консоли (SVGA), так и в среде X-Windows.

В операционной системе UNIX (в частности в Linux) существует одна особенность, которую следует учитывать при разработке программ, имеющих средства визуализации. Дело в том, что консоль кластера (главная машина кластера) не является персональным компьютером в полном смысле этого слова. Скорее всего, пользователь, проводящий численное моделирование на кластере, не будет подключен к системе в течение всего времени счета задачи. Закрытие пользователем сеанса работы (logout) без снятия задачи со счета или приостановки задачи возможно только в том случае, когда задача выполняется в режиме демона. То есть у задачи отсутствует как консольный, так и графический вывод. Другими словами, задача выполняется в системе в фоновом режиме. В случае с параллельной виртуальной машиной (PVM – Parallel Virtual Machine) это означает, во-первых, отсутствие постоянно действующего графического интерфейса задачи, и, во-вторых, программа запущена командой `spawn` без перенаправления вывода в `stdout` на консоль. То есть использовать "`spawn -> myprog`" нельзя.

Когда программа запускается командой `spawn` без перенаправления, весь консольный вывод программы записывается в log-файл системы PVM (например, в `/tmp/pvml.0`). Таким образом, пользователь, запустив программу, может безболезненно закончить сеанс работы с консолью, программа при этом будет продолжать работу. Мониторинг получаемых данных можно в любой момент осуществить с помощью стандартной команды операционной системы `tail`.

```
tail -f /tmp/pvml.0
```

Запустив эту команду, вы можете в реальном времени отслеживать изменение log-файла. Текст, добавляемый программой в конец log-файла, будет тут же отображаться у вас на текстовой консоли.

Несколько сложнее обстоит дело с процессом визуального отображения данных. Закрытие пользовательского сеанса в худшем случае приведет к аварийному останову программы, в лучшем к закрытию графического окна и к невозможности восстановить это окно при следующих сеансах работы.

Решить эту проблему можно двумя разными способами. Первый способ заключается в разнесении процессов счета и визуализации на две отдельные задачи. Первая запускается на исполнение средствами параллельной виртуальной машины, вторая является обычной пользовательской задачей, которая будет работать только тогда, когда это необходимо. В этом варианте графическая программа должна как-то получать от основной программы данные, подлежащие визуализации. Наиболее простым способом общения этих двух программ является их связь через файл, в который параллельная программа будет периодически (например, после каждой итерации) записывать массив данных. Графическая программа, читая этот файл, будет в цикле прорисовывать картинку, в соответствии с полученными данными.

Однако, использовать для подобного процесса обычные файлы нельзя, поскольку могут возникнуть непреодолимые сложности, связанные с синхронизацией операций чтения и записи в файл. Может произойти ситуация, когда графическая программа начала читать очередной блок данных, а программа счета в это время еще не закончила писать данные в файл. В этом случае весьма вероятен сбой. Вторая сложность заключается в объеме передаваемых через файл данных. Расчетная программа на каждой итерации

может генерировать достаточно большое количество данных, которые требуется визуализировать. При ограниченном объеме жестких дисков в короткое время файловая система может переполниться, что не приведет ни к чему хорошему.

В операционных системах UNIX имеется такое понятие, как «именованные каналы» или FIFO (first in, fisrt out). Другими словами именованный канал является буфером. Данные, которые в него попадают при записи, могут быть прочитаны из него в той же последовательности, причем прочитанные данные автоматически удаляются из FIFO. Поскольку в UNIX все, начиная с текстовых файлов и кончая сокетами и процессором, для пользователя является файлами, к которым можно обратиться обычными средствами файлового доступа, определенными в используемом языке программирования, то и именованные каналы FIFO тоже суть файлы. Используя FIFO для обмена данными, автоматически решается проблема синхронизации чтения/записи для программ и, кроме того, прочитанные и обработанные данные будут автоматически удалены из файловой системы, предотвращая ее переполнение.

Как уже было сказано, с точки зрения программиста, работа с файлами FIFO ничем не отличается от работы с обыкновенными файлами. Отличие заключается в методе создания такого файла. Для создания FIFO следует воспользоваться командой операционной системы `mkfifo`. Для этого в каталоге, где будет располагаться именованный канал, необходимо выполнить следующую команду:

```
mkfifo my_fifo_data
```

Параметром команды является имя создаваемого файла FIFO.

После этого в расчетной части программы, точнее в том её процессе, который будет исполняться на консоли кластера, программист должен организовать периодический сброс в файл FIFO необходимых для визуализации процесса блоков данных. Открытие файла FIFO в режиме write only и его закрытие может выполняться соответственно в самом начале и в самом конце программы. Оператор записи в файл будет находиться в теле цикла. В программе визуализации именованный канал также открывается и закрывается вне тела цикла. Открытие файла происходит в режиме read only.

Особенностью чтения данных из FIFO является отсутствие для программиста необходимости каким-либо образом организовывать процесс ожидания поступления новой порции данных. Если в программе управление передано оператору чтения до фактического появления в FIFO новой порции данных, выполнение программы автоматически приостанавливается до того момента, когда эти данные появятся в FIFO.

Следует, однако, отметить, что длительное отсутствие пользователя в системе и, соответственно, отсутствие в системе работающей программы визуализации, которая читает данные из FIFO, приводит к постепенному росту размера файла FIFO и, в конце концов, переполнению, файловой системы, как это происходило бы с обычным файлом.

Рассмотрим второй способ организации процесса визуализации данных. Поскольку с точки зрения операционной системы процесс, запущенный на узле параллельной виртуальной машины, является полноправной пользовательской задачей, то такая задача естественно может иметь графический интерфейс. Другими словами программист имеет

возможность заставить материнский процесс параллельной задачи открыть графическое окно и что-либо в этом окне нарисовать. Если выбран такой способ визуализации, то нет необходимости использовать ресурсы файловой системы, и тогда исчезают проблемы, связанные с файловым доступом, о которых мы говорили выше.

Однако задача, имеющая графический интерфейс, не работает в режиме демона (фоновом режиме) и, соответственно, нет возможности безболезненно для этой задачи закрыть пользовательскую сессию (logout). Проблема эта решаемая. Поскольку закрытие графического окна у задачи переводит ее в фоновый режим, можно заставить нашу программу визуализировать данные только тогда, когда мы этого хотим. Вопрос в том, как сообщить программе, что ей надо открыть графический интерфейс и показать нам, что она нам такое насчитала. Самый простой способ – это использовать файл-триггер, наличие которого будет сигнализировать программе, что графический интерфейс требуется, отсутствие же которого – есть сигнал программе закрыть графическое окно, если оно открыто. Наиболее просто управлять таким файл-триггером можно с помощью следующего простого скрипта:

```
#!/bin/sh
#
if [ ! -e ~/trigger.dat ]; then
   touch ~/trigger.dat
fi
SHOW_FLAG=`cat ~/trigger.dat`
if [ "${SHOW_FLAG}" = "0" ]; then
   echo -n 1 > ~/trigger.dat
   else
   echo -n 0 > ~/trigger.dat
fi
```

Этот скрипт проверяет содержимое файла `trigger.dat`, находящегося в домашнем каталоге пользователя (при необходимости создавая этот файл), и меняет его содержимое в циклическом порядке: `"0"` меняет на `"1"` и наоборот. Программа должна периодически читать этот файл, и в зависимости от прочитанных данных открывать или закрывать графическое окно.

Перейдем непосредственно к способам создания в программе графического окна и прорисовки в нем изображения. Как уже говорилось ранее, инструментов для работы с графикой существует огромное количество. Но существует одна очень простая в установке и использовании библиотека: `G2`. Эта библиотека имеет ещё то преимущество, что может использоваться как в программах, написанных на *C*, так и в программах на *Fortran*. Получить эту библиотеку можно на сайте разработчиков по адресу `http://sourceforge.net/projects/g2/` или скачать со странички по адресу `http://cluster.linux-ekb.info/download.php`. На момент редактирования методического пособия по лабораторным работам текущая версия библиотеки была 0.72. Библиотека распространяется в исходных кодах. Данная библиотека может с одинаковой легкостью использоваться как для прорисовки графики в окне X-Windows, так и для создания аналогичной графики в виде файлов `.png` или `.jpg`. Последнее представляет интерес для создания последовательности слайдов из которых при желании можно

сделать презентацию или фильм. Так же с помощью этой библиотеки аналогично созданию графических файлов или прорисовки графики в окне создавать файлы в формате Enchanced PostScript, подготовленные для печати на любом принтере, понимающем PostScript или, если печать ведется в Linux, вообще на любом принтере.

Трансляция и выполнение MPI-программ

Для трансляции и компоновки программ на языке *C* – команда **gcc**. Пример применения команды:

```
gcc -I../src -I../src/X11 -I/usr/local/include source.c -o
        program -L.. -lg2 -L/usr/local/lib -lm -lX11
```

где **source.c** – имя файла с исходным кодом программы, **program** – желаемое имя исполняемого файла программы.

Программа запускается командой:

```
./program [ключи и аргументы программы]
```

где **program** – имя исполняемого файла программы.

Лабораторная работа

Задание 1

Установите библиотеку `G2` в систему PelicanHPC. Для этого необходимо сначала распаковать архив с исходными кодами в каком-либо каталоге, например в домашнем: `/home/user/g2-0.72`.

После этого, для установки библиотеки, вы должны воспользоваться последовательностью команд:

```
sudo ./configure
sudo make depend
sudo make
sudo make install
```

Задание 2

Напишите демонстрационную программу на *C*. Откомпилируйте, запустите и посмотрите результат работы. Сделайте выводы.

Для этого рассмотрим практические вопросы программирования с использованием библиотеки `G2` на языке *C*. Подробно рассматривать все функции, которыми вы можете пользоваться при создании графических изображений нецелесообразно, описания функций вы всегда можете посмотреть в прилагаемой к библиотеке документации. Вместо этого приведем два примера программ, которые в окне X-Windows размером 640x640 пикселей рисуют распределение некоей функции, и кратко опишем использованные в программе функции.

Для конкретики предположим, что мы считаем некоторую двумерную газодинамическую задачу. Разностная сетка у нас имеет размер 640x640. По окончании каждой итерации счета мы получаем распределение плотности по пространству. Для

отображения на картинке величины плотности в каждой ячейки нашей разностной сетки мы будем использовать 255 градаций серого цвета. Минимальная плотность будет рисоваться черным цветом, максимальная – белым. Конечно, саму программу расчета дифференциальных уравнений мы писать не будем, вместо этого в том месте программы, где мы должны получить массив плотностей, запишем простой цикл, в котором этот массив будет заполнен неким осмысленным, но произвольным образом.

Итак, исходный код программы на языке *C*:

Листинг 1. Демонстрационная программа на языке *C*:

```
#include <g2.h>
#include <g2_X11.h>

main()
{
   int d,i,x,y;
   //массив плотностей
   double q[640][640];
   //массив соответствующих цветов
   int colors[640][640];
   //специальный массив, в котором хранятся оттенки серого цвета
   int grays[255];

   //откроем графическое окно размером 640x640
   d=g2_open_X11(640,640);

   //подготавливаем 255 возможных оттенков серого
   for(i=0;i<255;i++)
      grays[i]=g2_ink(d, (double) (i/254.0), (double) (i/254.0),
(double) (i/254.0));

   ////////////////////////////////////////////////////////////
   //подготавливаем массив плотности (произвольным образом)
   //вместо чтения из fifo-файла или из pvm-сообщения
   //заполняем массив вручную
   //в реальной задаче вместо этого куска кода будет
   //стоять один оператор чтения данных из файла или
   //операторы приема сообщения и распаковки его в массив
   for(x=0;x<640;x++)
   {
      for(y=0;y<640;y++)
      {
         q[x][y]=sqrt(abs((300*300)-(x*y)));
      }
   }
   ////////////////////////////////////////////////////////////

   //вычисляем максимум и минимум плотности
   double min=q[0][0]; double max=q[0][0];
   for(x=0;x<640;x++)
```

```
    {
        for(y=0;y<640;y++)
        {
            if(min>q[x][y]) min=q[x][y];
            if(max<q[x][y]) max=q[x][y];
        }
    }

    //вычисляем цвета пикселей
    for(x=0;x<640;x++)
    {
        for(y=0;y<640;y++)
        {
            //конформно отражаем распределение плотностей
            //к диапазону 0-254 и в качестве цвета пикселя
            //берем целую часть от полученной величины
            int c=(int) ((254*(q[x][y]-min))/(max-min));
            colors[x][y]=grays[c];
        }
    }

    //рисуем распределение плотности в графическом окне X-Windows
    g2_image(d,0.0,0.0,640,640,&colors[0][0]);

    //рисуем белый прямоугольник в котором расположим текст
    int color = g2_ink(d,1.0,1.0,1.0);
    g2_pen(d,color);
    g2_filled_rectangle(d,15.0,15.0,80.0,35.0);

    //рисуем черным цветом подпись к картинке в
    //нарисованном белом прямоугольнике
    color = g2_ink(d,0.0,0.0,0.0);
    g2_pen(d,color);
    g2_string(d,20.0,20.0,"Test field");

    //немножко поспим чтобы можно было полюбоваться
    //полученной картинкой
    sleep(10);

    //закроем графическое окно
    g2_close(d);
}
```

Результатом работы программы является картинка, показанная на рисунке 1.

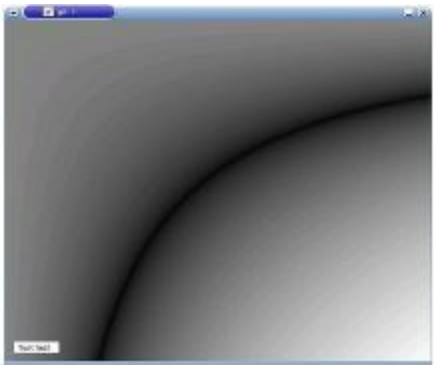

Рисунок 1 – Результат выполнения демонстрационной программы

В том случае, когда прорисовкой графики занимается материнский процесс параллельной задачи, алгоритм по которому построен этот процесс может быть таким:

```
<инициализируем параллельный процесс>
do <итерация>=1 to N
   <обмениваемся данными с дочерними процессами>
   if <флаг визуализации> = 1 then
      if <графическое окно не открыто> then
         <открываем графическое окно>
      endif
      <подготавливаем данные для прорисовки>
      <рисуем картинку>
   else
      if <графическое окно запущено> then
         <закрываем графическое окно>
      endif
   endif
enddo
<заканчиваем работу параллельного процесса>
```

Теперь опишем вкратце те функции, которые использовали в тестовой программе.

Функция открывает X11 окно шириной `width` пикселей и высотой `height` пикселей, возвращает идентификатор устройства `dev` вывода, который используется в других функциях:

```
int dev = g2_open_X11(int width, int height)
```

Функция закрывает графическое устройство с идентификатором `dev` открытое предыдущей функцией:

```
void g2_close(int dev)
```

Функция рисует точку (пиксель) цветом, установленным функцией `g2_pen`:

```
void g2_plot(int dev, double x, double y)
```

Функция рисует прямоугольник с координатами левого нижнего угла (`x1`,`y1`) и координатами правого верхнего угла (`x2`,`y2`), после чего заливает этот прямоугольник цветом, установленным функцией `g2_pen`.

```
void g2_filled_rectangle(int dev, double x1, double y1,
            double x2, double y2)
```

Функция рисует текст, заданный переменной `text` позиция начала текста определяется координатами (`x`,`y`) цвет текста устанавливается функцией `g2_pen`:

```
void g2_string(int dev, double x, double y, char *text)
```

Функция заполняет прямоугольную область с координатами левого нижнего угла (`x`,`y`), шириной `x_size` и высотой `y_size` цвета пикселей этой области задаются в двумерном массиве `pens` размерностью (`x_size`,`y_size`):

```
void g2_image(int dev, double x, double y, int x_size,
              int y_size, int *pens)
```

Функция создает новый цвет определяемый RGB параметрами `red`,`green`,`blue` значения этих параметров – действительные числа в диапазоне `0..1` функция возвращает индекс нового цвета `color`:

```
int color=g2_ink(int dev, double red, double green, double blue)
```

Функция устанавливает текущий цвет для рисования; цвет определяется индексом цвета `color`, который получен с помощью функции `g2_ink`:

```
void g2_pen(int dev, int color)
```

Задание 3

Выполните задание в соответствие с заданным вариантом.

1) Напишите программу поиска 500 простых чисел с использованием парадигмы MPI. Используя библиотеку G2, постройте график распределения простых чисел. Например, график распределения простых чисел (для девяти первых простых чисел) может выглядеть примерно так, как показано на рисунке 2.

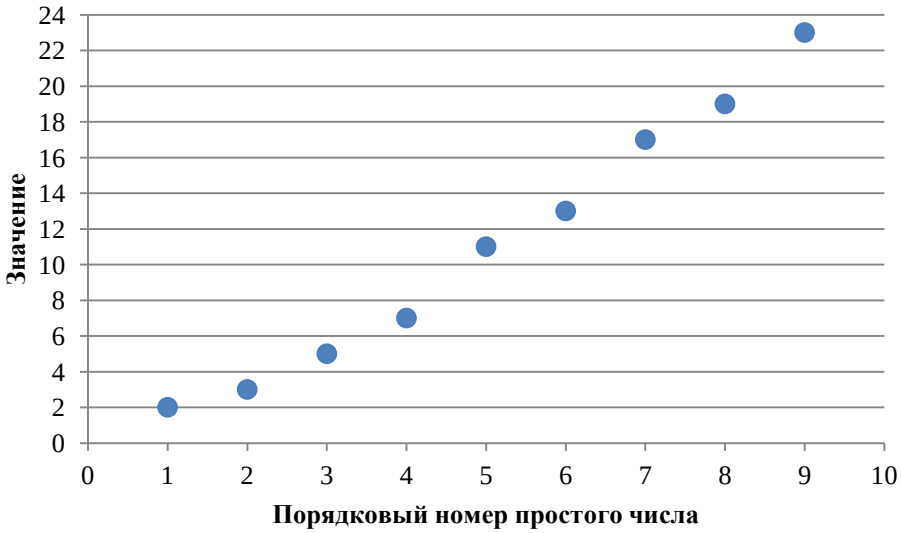

Рисунок 2 – График распределения девяти простых чисел

2) Найти и показать клетки на шахматной доске, которые не бьются ферзем.
3) Найти и показать клетки на шахматной доске, которые бьются ферзем.
4) Вычислите и покажите на стандартной шахматной доске 8x8 возможное количество ходов из данной позиции конём (позиция задаётся пользователем).

5) Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого. Найдите количество решений и изобразите одно из них по выбору пользователя.
6) Заполните массив из 100 элементов с помощью генератора случайных чисел. До тех пор, пока числа не выстроятся в порядке возрастания или убывания генерируйте новые значения элементов массива. Когда условие будет выполнено, выведите содержимое массива на экран, выделив с помощью градиента (например, самое маленькое число – зелёным, самое большое – красным цветом, а все промежуточные значения оттенками жёлтого) значения элементов массива.
7) Заполните двумерный массив 10 на 10 элементов с помощью генератора случайных чисел. Изобразите содержимое массива на экране при помощи геометрических фигур (например, круг или квадрат), размер которых пропорционален числу, находящемуся на данном месте в массиве.
8) Изобразите график зависимости времени поиска простого числа от его порядкового номера для 500 простых чисел.
9) Найти как можно больше простых чисел, которые имеют порядковый номер, являющийся также простым числом. Например, в таблице ниже выделены простые числа, удовлетворяющие условиям задачи:

Порядковый номер	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
Простое число	**2**	**3**	**5**	7	**11**	13	**17**	19	23	29	**31**	37	**41**	43	47	53	**59**	61	**67**	71

Результаты работы программы сохранить в файл (список простых чисел, удовлетворяющих условиям задачи), а на экран вывести максимальное простое число, которое удалось найти в соответствии с условием задачи, при этом время работы программы не должно превышать 60 секунд. Работу таймера изобразить с помощью графической библиотеки.
10) Найти как можно больше простых чисел, состоящих из нулей и единиц. Например, в пределах первых десяти миллионов, содержатся следующие числа: 11, 101, 10111, 101111, 1011001, 1100101 и т.д. Каждое найденное число отобразить с помощью графической библиотеки, например, выделив нули и единицы разным цветом. Также можно визуализировать процесс поиска чисел. Время работы программы не должно превышать 120 секунд
11) Необходимо вставить числа вместо знаков вопроса, чтобы звезда заработала (см. рисунок 3). Т.е. должны выполняться логические соотношения, указанные над линиями. Например, если в середину звезды вставить двоичное число `110011`, а в правую нижнюю ветвь число `110111`, то соотношение `110011 OR 110111 = 110111` выполняется, но в данном случае невозможно будет подобрать значения к другим веткам.

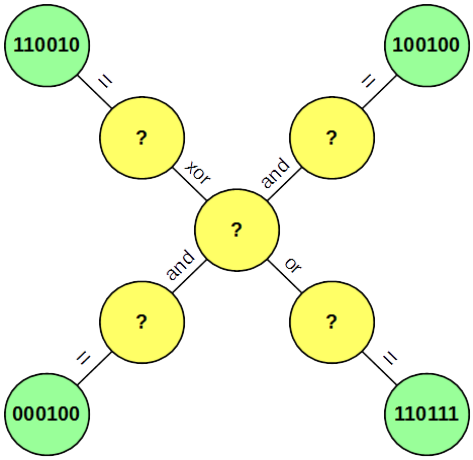

Рисунок 3 – Логическая звезда

Напишите программу, находящую все множество возможных значений для этой звезды, и изобразите одно из решений по выбору пользователя.

Вопросы

1) Какие графические библиотеки вы знаете?
2) Как с помощью библиотеки G2 получить результат в виде графического файла `.png` или `.jpg`?
3) Что нужно изменить в программе из задания 2, чтобы получить рисунок не в отдельном окне, а в консоли?