УДК 681.3.06:62-507

РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ КЛЕТОЧНЫХ АВТОМАТОВ ПОСРЕДСТВОМ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ CAME&L (Часть II)

Л. А. Наумов,
аспирант
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Работа представляет собой введение в программирование для пакета CAME&L посредством библиотеки CADLib. Описываются основы решения задач с помощью рассматриваемого программного обеспечения, а именно – создание пользовательских правил для клеточных автоматов. На примере игры «Жизнь» демонстрируется разработка простейших решений, использование зональной оптимизации, поддержка многопроцессорной и кластерной вычислительных систем, а также – обобщенных координат. Приводится пример решения физической задачи, уравнения теплопроводности.

This work represents the introduction into programming for CAME&L software by means of CADLib library. Bases of building tasks' solutions with the help of considered software, namely the creation of user rules for cellular automata, are described. Here five variants of «Game of Life» are introduced: plain one, variant with zonal optimization, implementations for multiprocessor and cluster computing systems and for generalized coordinates. Moreover the solution of physical problem, thermal conductivity equation, is described.

Реализация игры «Жизнь»

Клеточный автомат Джона Хортона Конвея «Жизнь» [1–3] настолько известен, что нет необходимости приводить словесное описание его правил. Данный раздел посвящен пяти вариантам реализации этого автомата: простейшей реализации «в лоб», реализации с зональной оптимизацией, реализации для многопроцессорной и кластерной вычислительных систем, для обобщенных координат [4].

Простейшая реализация. Приведем реализацию игры «Жизнь» для однопроцессорной системы, без оптимизации, для привычной картезианской метрики.

При этом необходимо разработать класс, реализующий соответствующий компонент правил. Назовем его `CALifeRules`. В качестве решетки автомата можно выбрать любую из стандартных двумерных решеток (наиболее привычна квадратная, стандартный компонент «Square Basic Grid»). В качестве метрики – картезианские координаты (стандартный компонент «Cartesians»), а хранилища данных – хранилище для булевых величин (стандартный компонент «Booleans for Cartesians»).

Необходимо отметить, что перед проведением эксперимента потребуется включить хранение данных на двух последовательных шагах (параметр «Double» хранилища перевести в значение `true`). В результате внутри компонента будут созданы две копии структуры данных для размещения состояния решетки. Значения будут получаться из одной из них, а помещаться – в другую. После каждого шага структуры будут меняться местами. Это обеспечит независимость результирующего состояния решетки от порядка перебора клеток – неотъемлемое свойство «классических» клеточных автоматов.

Введем в разрабатываемом компоненте правил два набора по 13 параметров булевого типа (массивы `p_bBorn` и `p_bDie`), которые позволят определять функцию переходов автомата. Если значение параметра `p_bBorn[i]` равно `true`, то мертвая клетка оживает при наличии *i* живых соседей. Если значение параметра `p_bDie[i]` равно `true`, то живая клетка умирает при наличии *i* живых соседей. Значение *i* лежит от нуля до двенадцати (наибольшего числа непосредственных соседей на треугольной решетке), однако на квадратной решетке будут использованы лишь первые девять из них (от нуля до восьми).

Кроме того, введем целочисленный анализируемый параметр p_iAlive, который позволит наблюдать за динамикой изменения количества живых клеток на решетке.

Для сопоставления компоненту некой иконки необходимо создать соответствующий ресурс. Пусть его идентификатор будет IDI_ICON.

Объявление класса CALifeRules может иметь следующий вид:

```
class CALifeRules:public CARules
{
public:
    // Конструктор и деструктор
    CALifeRules();
    virtual ~CALifeRules();

    // Определение имени, описания, иконки и условий совместимости
    COMPONENT_NAME(Game of Life (plain))
    COMPONENT_INFO(Conways game of Life rules (plain variant))
    COMPONENT_ICON(IDI_ICON)
    COMPONENT_REQUIRES(Data.bool.*&Metrics.2D.cartesian.*)

    // Следующие функций класса CARules будут переопределены
    virtual bool Initialize();
    virtual bool SubCompute(Zone& z);

public:
    // Параметры
    ParamBool* p_bBorn[13];
    ParamBool* p_bDie[13];

    // Анализируемый параметр
    ParamInt p_iAlive;

public:
    // Карта параметров
    PARAMETERS_COUNT(26)
    BEGIN_PARAMETERS
        PARAMETER(0,p_bBorn[0])
        PARAMETER(1,p_bBorn[1])
        PARAMETER(2,p_bBorn[2])
        PARAMETER(3,p_bBorn[3])
        PARAMETER(4,p_bBorn[4])
        PARAMETER(5,p_bBorn[5])
        PARAMETER(6,p_bBorn[6])
        PARAMETER(7,p_bBorn[7])
        PARAMETER(8,p_bBorn[8])
        PARAMETER(9,p_bBorn[9])
        PARAMETER(10,p_bBorn[10])
        PARAMETER(11,p_bBorn[11])
        PARAMETER(12,p_bBorn[12])
        PARAMETER(13,p_bDie[0])
        PARAMETER(14,p_bDie[1])
        PARAMETER(15,p_bDie[2])
        PARAMETER(16,p_bDie[3])
        PARAMETER(17,p_bDie[4])
        PARAMETER(18,p_bDie[5])
        PARAMETER(19,p_bDie[6])
        PARAMETER(20,p_bDie[7])
        PARAMETER(21,p_bDie[8])
        PARAMETER(22,p_bDie[9])
        PARAMETER(23,p_bDie[10])
        PARAMETER(24,p_bDie[11])
        PARAMETER(25,p_bDie[12])
    END_PARAMETERS

    // Карта анализируемых параметров
    APARAMETERS_COUNT(1)
    BEGIN_APARAMETERS
        PARAMETER(0,&p_iAlive)
    END_APARAMETERS

private:
    // Вспомогательная переменная для вычисления значения p_iAlive
    int iq;
};
```

Три необходимых каждой библиотеке компонента функции можно создать с помощью описанных выше макроопределений:

```
COMPATIBLE_RULES(1.0)
RULES_COMPONENT(CALifeRules)
```

Основное назначение конструктора и деструктора компонента состоит в создании, инициализации, а также удалении параметров. В данном случае они будут иметь вид

```
CALifeRules::CALifeRules():
      p_iAlive("Alive","Amount of alive cells",0)
{
      p_bBorn[0]=new ParamBool("Born if 0","Is dead cell to become alive if it has 0 alive neighbors",false,this);
      p_bBorn[1]=new ParamBool("Born if 1","Is dead cell to become alive if it has 1 alive neighbors",false,this);
      p_bBorn[2]=new ParamBool("Born if 2","Is dead cell to become alive if it has 2 alive neighbors",false,this);
      p_bBorn[3]=new ParamBool("Born if 3","Is dead cell to become alive if it has 3 alive neighbors",true,this);
      p_bBorn[4]=new ParamBool("Born if 4","Is dead cell to become alive if it has 4 alive neighbors",false,this);
      p_bBorn[5]=new ParamBool("Born if 5","Is dead cell to become alive if it has 5 alive neighbors",false,this);
      p_bBorn[6]=new ParamBool("Born if 6","Is dead cell to become alive if it has 6 alive neighbors",false,this);
      p_bBorn[7]=new ParamBool("Born if 7","Is dead cell to become alive if it has 7 alive neighbors",false,this);
      p_bBorn[8]=new ParamBool("Born if 8","Is dead cell to become alive if it has 8 alive neighbors",false,this);
      p_bBorn[9]=new ParamBool("Born if 9","Is dead cell to become alive if it has 9 alive neighbors",false,this);
      p_bBorn[10]=new ParamBool("Born if 10","Is dead cell to become alive if it has 10 alive neighbors",false,this);
      p_bBorn[11]=new ParamBool("Born if 11","Is dead cell to become alive if it has 11 alive neighbors",false,this);
      p_bBorn[12]=new ParamBool("Born if 12","Is dead cell to become alive if it has 12 alive neighbors",false,this);

      p_bDie[0]=new ParamBool("Die if 0","Is alive cell to become dead if it has 0 alive neighbors",true,this);
      p_bDie[1]=new ParamBool("Die if 1","Is alive cell to become dead if it has 1 alive neighbors",true,this);
      p_bDie[2]=new ParamBool("Die if 2","Is alive cell to become dead if it has 2 alive neighbors",false,this);
      p_bDie[3]=new ParamBool("Die if 3","Is alive cell to become dead if it has 3 alive neighbors",false,this);
      p_bDie[4]=new ParamBool("Die if 4","Is alive cell to become dead if it has 4 alive neighbors",true,this);
      p_bDie[5]=new ParamBool("Die if 5","Is alive cell to become dead if it has 5 alive neighbors",true,this);
      p_bDie[6]=new ParamBool("Die if 6","Is alive cell to become dead if it has 6 alive neighbors",true,this);
      p_bDie[7]=new ParamBool("Die if 7","Is alive cell to become dead if it has 7 alive neighbors",true,this);
      p_bDie[8]=new ParamBool("Die if 8","Is alive cell to become dead if it has 8 alive neighbors",true,this);
      p_bDie[9]=new ParamBool("Die if 9","Is alive cell to become dead if it has 9 alive neighbors",true,this);
      p_bDie[10]=new ParamBool("Die if 10","Is alive cell to become dead if it has 10 alive neighbors",true,this);
      p_bDie[11]=new ParamBool("Die if 11","Is alive cell to become dead if it has 11 alive neighbors",true,this);
      p_bDie[12]=new ParamBool("Die if 12","Is alive cell to become dead if it has 12 alive neighbors",true,this);
```

```
      // Определение комментария, отображаемого в строке состояния
      sComment[1]="Game of Life";
}

CALifeRules::~CALifeRules()
{
      for(unsigned char b=0;b<13;b++) delete p_bBorn[b];
      for(b=0;b<13;b++) delete p_bDie[b];
}
```

Потребность в обработчике инициализации (перегрузке функции `Initialize`) возникает только из-за наличия анализируемого параметра, значение которого должно быть известно перед выполнением первой итерации. В результате эта функция примет вид

```
bool  CALifeRules::Initialize()
{
    // Инициализацию следует производить, только если
    //параметр анализируется
    if (!p_iAlive.IsAnalyzed()[2]) return true;

    DATUM[3](CACrtsBoolDatum);
    CACell c; // Текущая клетка
    iq=0;

    for(int i=(int)datum->z.a1; i<=(int)datum->z.b1; i++)
         for(int j=(int)datum->z.a2; j<=(int)datum->z.b2; j++)
       {
              // Получение идентификатора текущей клетки
              // по декартовым координатам
               c=GetUnion()->Metrics->ToCell(i,j,0);

              if (datum->Get(c)) iq++;
       }
    // Присваивание значения анализируемому параметру
    p_iAlive.Set(iq);

    return true;
}
```

Необходимо отметить, что переменная-член `Zone z` класса `CADatum` описывает зону, значения состояний клеток из которой содержатся в хранилище. Эта переменная используется в приведенной выше функции для организации перебора всех клеток в цикле.

Далее приводится основная функция, осуществляющая итерации. Как отмечалось выше, даже для однопроцессорной системы, не предусматривающей распараллеливания вычислений, реализацию шага лучше осуществлять в функции `SubCompute`. Пример такого решения:

```
bool  CALifeRules::SubCompute(Zone&  z)
{
    DATUM(CACrtsBoolDatum);

    CACell c; // Текущая клетка
    CACell neig[12]; // Массив для хранения всех соседей
    // Переменная, хранящая количество соседей
    unsigned short ncount=GET_METRICS[4]->GetNeighboursCount();
    unsigned char alive; // Количество живых соседей
                         // текущей клетки
    iq=0;

    for(CACell i=z.a1; i<=z.b1; i++)
         for(CACell j=z.a2; j<=z.b2; j++)
       {
            alive=0;

              // Получение идентификатора текущей клетки по декартовым
              // координатам
               c=GetUnion()->Metrics->ToCell(i,j,0);

              // Получение всех соседей и размещение их в массиве
               pUnion->Metrics->GetNeighbours(c,neig);

              // Подсчет числа живых соседей текущей клетки
              for (int k=0; k<ncount; k++) {
                    if (datum->Get(neig[k])) alive++;
             }

              // Применение функции переходов и определение значения
              // анализируемого параметра
             if (datum->Get(c)) {
                    datum->Set(c,!p_bDie[alive]->Get());
                   if (!p_bDie[alive]->Get()) iq++;
              } else {
                    datum->Set(c,p_bBorn[alive]->Get());
                   if (p_bBorn[alive]->Get()) iq++;
             }
       }
    // Присваивание значения анализируемому параметру
    p_iAlive.Set(iq);

    return true;
}
```

Таким образом, пример простейшей реализации клеточного автомата «Жизнь» построен. Однако следует отметить, что перед его использованием компонент необходимо скомпилировать и уставить в среду, для чего требуется выбрать в главном меню «Tools» | «Components Manager...» (или воспользоваться горячей клавишей <F9>), затем нажать в появившемся окне кнопку «Add...» и открыть файл библиотеки компонента.

В настоящей работе не будут обсуждаться такие вопросы, как включение необходимых заголовочных файлов и подключение библиотек, в силу громоздкости описания этой тематики, а также для того, чтобы не обижать читателей предположением, что они не смогут сделать этого самостоятельно.

В заключение отметим, что вычисление анализируемых параметров в процессе выполнения шага автомата позволяет существенно сэкономить вычислительное время и избавиться от повторных переборов всех клеток решетки.

Реализация с использованием зональной оптимизации. Понаблюдав за моделированием рассматриваемого клеточного автомата, нетрудно заметить, что почти всегда изменения состояний клеток про-

[1] Переменная `sComment` – член класса `CARules`.

[2] Функция `IsAnalyzed()` – член класса `Parameter`.

[3] Данное макроопределение создает локальную переменную `datum`, указатель на хранилище данных, с которым работает компонент правил. При этом он приводится к типу указателя на класс, передаваемый макроопределению в качестве параметра.

[4] Макроопределение `GET_METRICS` обеспечивает быстрый доступ к метрике, с которой работает данный компонент. Аналогично имеются макроопределения `GET_GRID` и `GET_DATUM`.

исходят лишь в некоторых локализованных областях. При этом большие зоны, состоящие сплошь из мертвых клеток, можно исключать из рассмотрения, так как на данном шаге жизнь в них зародиться не сможет. Отбрасывание таких областей будем называть зональной оптимизацией вычислений.

Для осуществления такой оптимизации в библиотеке CADLib имеется класс `OptZonal`. На каждой итерации объект этого класса запоминает клетки, состояние которых может повлиять на соседей на следующих итерациях (назовем их значимыми клетками). Пользуясь этой информацией, он формирует зоны, которые следует принимать в расчет при вычислениях на следующем шаге.

Приведем описание нескольких членов класса, которые будут использованы для реализации клеточного автомата «Жизнь» с зональной оптимизацией.

- `public OptZonal(unsigned int diffusion,unsigned int unite)` – конструктор. Параметр `diffusion` определяет величину «полей» зон, расстояние от крайних значимых клеток до границы зоны. Фактически, этот параметр должен быть равен радиусу окрестности клеток [4]. Параметр `unite` определяет наибольшее расстояние между клетками двух зон, при котором их следует объединять в одну.
- `public inline void Reset()` – функция инициализирует объект.
- `public void AddCell(CACell x,CACell y=0,CACell z=0)` – функция добавляет информацию о значимой клетке с соответствующими координатами вдоль трех осей.
- `public void NewStep()` – функция переключает указатель на рабочее хранилище информации о зонах между двумя внутренними хранилищами, проверяет, перекрываются ли зоны в них, и т. п. Она должна вызываться перед каждым новым шагом для того, чтобы объект данного класса вычислял границы зон для следующего шага.
- `public bool ReleaseZone(Zone& z)` – функция возвращает через параметр `z` границы зоны, которую следует обработать. Ее следует вызывать до тех пор, пока результат, возвращаемый самой функцией, не станет равным `false`.

Для того чтобы воспользоваться зональной оптимизацией, в решение, разработанное в подразд. «Простейшая реализация» (с. 30), необходимо внести ряд изменений.

Во-первых, потребуется добавить переменную – член класса, реализующего компонент правил, `OptZonal oOpt`. В конструкторе класса `CALifeRules` необходимо вызвать конструктор для этой переменной с соответствующими параметрами, например `oOpt(1,4)`, а также присвоить значение `true` переменной `bFinalizeIfChanged`[1]. В результате изменение состояние решетки вручную приведет к финалу эксперимента. Это необходимо для обеспечения целостности информации о зонах.

Во-вторых, в функции `Initialize` класса, реализующего правила, для обеспечения корректности информации об обновляемых зонах необходимо сделать обнуление второго буфера данных, если включено двойное хранилище. В результате функция примет вид

```
bool  CALifeRules::Initialize()
{
     // Инициализация оптимизатора
     oOpt.Reset();

     DATUM(CACrtsBoolDatum);
     CACell c; // Текущая клетка
     iq=0;

     for(CACell i=datum->z.a1; i<=datum->z.b1; i++)
          for(CACell j=datum->z.a2; j<=datum->z.b2; j++)
        {
               // Получение идентификатора текущей клетки
               // по декартовым координатам
                c=GetUnion()->Metrics->ToCell(i,j,0);

              if (datum->Get(c)) {
                iq++;
                   // Данная клетка живая, а следовательно -
                   // значимая
                  oOpt.AddCell(i,j);
            }
        }
     // Присвоение значения анализируемому параметру
     p_iAlive.Set(iq);

     // Обнуление второе буфера данных, если включено
     // двойное хранение
     if (datum->p_bDouble.Get()) {
          for(CACell i=datum->z.a1; i<=datum->z.b1; i++)
               for(CACell j=datum->z.a2; j<=datum->z.b2; j++)
           {
                    c=GetUnion()->Metrics->ToCell(i,j,0);
                  datum->Set(c,false);
           }
    }

     return true;
}
```

В-третьих, в функции `SubCompute` необходимо добавить вызов функции `oOpt.NewStep` и производить вычисления только для тех зон, которые возвращаются функцией `oOpt.ReleaseZone`. Кроме этого, как и в обработчике инициализации, необходимо вызывать функцию `oOpt.AddCell` для всех оживших клеток.

В результате этих изменений производительность вычислений повысится, в среднем, на порядок.

Реализация для многопроцессорной системы. Идея выполнения клеточного автомата «Жизнь» на многопроцессорной вычислительной системе заключается в создании большого числа потоков [5], каждый из которых будет выполнять функцию `SubCompute` для некоторой части пространства. Потоки, в свою очередь, распределяются между процессорами. Итерация

[1] Переменная `bFinalizeIfChanged` – член класса `CARules`.

считается законченной, когда завершились вычисления во всех потоках.

Для того чтобы рассматриваемый клеточный автомат выполнялся на многопроцессорной системе, в решение, разработанное в подразд. «Простейшая реализация», необходимо внести ряд изменений.

Во-первых, удалить из функции SubCompute обнуление временной переменной iq, а также присвоить нового значение анализируемому параметру p_iAlive.

Во-вторых, реализовать функцию Compute следующим образом (все используемые в данном фрагменте кода макроопределения реализованы в библиотеке CADLib):

```
bool  CALifeRules::Compute()
{
      DATUM(CACrtsBoolDatum);

      // Создание счетчиков стартовавших и завершившихся
      // потоков
      COUNTER(started);
      COUNTER(finished);

      // Инициализация будущего значения анализируемого параметра
     iq=0;

      // Разделение задачи на подзадачи для разных потоков
      MULTIFOR2(k1,4*(int)pNetwork->GetThis()->uCPUsCount[1],
            datum->z.a1,datum->z.b1,
            k2,4*(int)pNetwork->GetThis()->uCPUsCount,
            datum->z.a2,datum->z.b2,started,finished);
      // Возможен и такой вариант:
      //  MULTIFOR2_FIXED(k1,100,datum->z.a1,datum->z.b1,
      //        k2,100,datum->z.a2,datum->z.b2,started,finished);

      // Ожидание совпадения счетчиков. Проверка каждые
      // 20 миллисекунд
      WAIT_COUNTER(started,finished,20);

      // Присвоение значения анализируемому параметру
      p_iAlive.Set(iq);

      return true;
}
```

Комментарии делают ясным назначение каждой строки приведенной выше функции. Отдельно рассмотреть имеет смысл только макроопределения MULTIFOR2 и MULTIFOR2_FIXED. Запишем их в общем виде.

Макроопределение MULTIFORn(k1, count1, a1, b1,..., kn, countn, an, bn, started, finished) осуществляет деление n-мерной задачи (n лежит от одного до трех) на подзадачи, причем отрезок [ai, bi] вдоль i-й оси делится на counti частей. Параметр ki определяет имя создаваемой локальной переменной цикла вдоль i-й оси (i лежит от одного до n). Таким образом, для $n = 3$ будет создано count1*count2*count3 потоков. Последние два параметра передают названия счетчиков стартовавших и завершившихся потоков соответственно. Использование двух различных счетчиков (вместо одного) оправдывается рядом весомых соображений, обсуждение которых выходит за рамки настоящей работы.

Макроопределение MULTIFORn_FIXED(k1, size1, a1, b1,..., kn, sizen, an, bn, started, finished) осуществляет деление n-мерной задачи (n лежит от одного до трех) на подзадачи, причем отрезок [ai, bi] вдоль i-й оси делится на отрезки по sizei клеток (последний из них может быть меньше). Параметр ki определяет имя создаваемой локальной переменной цикла вдоль i-й оси (i лежит от одного до n). Таким образом, для $n = 3$ будет создано ((b1-a1+1)/size1)*((b2-a2+1)/size2)*((b3-a3+1)/size3) потоков (при делении здесь необходимо округлять результаты «вверх»). Последние два параметра передают названия счетчиков стартовавших и завершившихся потоков соответственно.

Реализация для вычислительного кластера. Для выполнения клеточного автомата в вычислительном кластере на всех машинах, входящих в него, должна быть запущена среда из пакета CAME&L. Пусть на одной из машин (назовем ее главной) создается эксперимент (главный эксперимент), в процессе которого будут выполняться кластерные вычисления.

При выполнении инициализации эксперимента на главной машине всем остальным (или не всем, в зависимости от стратегии разделения задачи) посылается команда: создать вспомогательный эксперимент, который представляет собой часть главного, подзадачу. Причем состояние фрагмента решетки для данной подзадачи отправляется с учетом «полей», зон, которые не будут обрабатываться, но состояния клеток из которых могут потребоваться для вычисления новых значений для клеток основной области.

Каждый шаг на главной машине состоит в отправлении команд всем участникам эксперимента обновить поля (главная машина сообщает каждой, кто владеет информацией о состоянии клеток их полей) и произвести итерацию (шаг во вспомогательных экспериментах).

Необходимо отметить, что каждый эксперимент идентифицируется по так называемому дескриптору, в который входят строковое имя эксперимента (для его изменения можно выбрать в меню среды «Modeling» | «Change Rules ID...» или воспользоваться «горячим» сочетанием клавиш <Shift>+<F12>), адрес главной машины эксперимента и его номер. Данный дескриптор должен быть уникальным, чтобы однозначно определять эксперимент. Неповторимость строкового имени легко обеспечить на каждой машине в отдельности. В результате обеспечивается уникальность пары имени и адреса главной машины. Номер используется при вычислениях в кластере. Для главного эксперимента он равен нулю. Все вспомогательные эксперименты получают уникальные номера последовательно, начиная с единицы.

[1] Переменная pNetwork->GetThis()->uCPUsCount хранит число процессоров на данной машине.

Когда главная машина сочтет нужным, она может запросить у остальных состояние их подрешеток (исключая поля). В результате состояние всей решетки будет «собрано» на одной машине. После этого можно продолжить эксперимент или прекратить его.

Реализация автомата «Жизнь» для вычислительного кластера, как и для многопроцессорной системы, не сильно отличается от решения, приведенного в подразд. «Простейшая реализация». Потребуется внести следующие изменения.

Для удобства введем еще один параметр, p_iGather, целое число, определяющее количество шагов, по истечении которого необходимо собрать состояние всей решетки на главной машине. В случае, если он равен нулю, сбор не осуществляется. В конструкторе класса CALifeRules необходимо будет вызвать конструктор этого параметра, а также, как и в подразд. «Реализация с использованием зональной оптимизации» (с. 32), присвоить значение true переменной bFinalizeIfChanged для обеспечения целостности данных.

Функция SubCompute останется неизменной, однако функция Initialize существенно поменяется. Кроме того, потребуется определить функции Compute и Finalize. Три новые функции будут иметь следующий вид:

```
bool  CALifeRules::Initialize()
{
      // Проверка того, работает ли сетевой интерфейс
      if (!pNetwork->IsWorking()) return false;
      // Разделение задачи между машинами
      CreateCluster(1,1,true,2);
      // Ожидание, пока все участники кластера
      // подтвердят создание экспериментов и готовность
      return WaitForClusterReady();
}

void  CALifeRules::Finalize()
{
      // Удаление экспериментов на всех участниках
      DestroyCluster();
}

bool  CALifeRules::Compute()
{
      // Осуществление итерации
      return ComputeCluster(p_iGather.Get() &&
            (Step+1-pCluster->Step>=p_iGather.Get()));
}
```

Как видно из фрагмента приведенного кода, кластерные вычисления организуются с помощью пакета CAME&L чрезвычайно просто ввиду наличия богатого набора средств, содержащихся в классе CARules.

Опишем две переменные и две функции, члены класса CARules, использованные выше.

• public Network* pNetwork – переменная хранит указатель на описание сетевого контекста и интерфейса рабочей станции.

• public Cluster* pCluster – переменная хранит указатель на менеджер кластера.

• public virtual unsigned int CreateCluster(unsigned int overlap, unsigned char type, bool usethis, int remoteness) – функция разделяет задачу на подзадачи, отбирает машины, участвующие в эксперименте, и рассылает команды для создания вспомогательных экспериментов. Функция имеет следующие параметры:

unsigned int overlap – определяет величину полей;

unsigned char type – определяет способ разделения задачи. В данный момент поддерживаются следующие значения:

- 0 – отладочный способ, делящий решетку на вертикальные полосы, количество которых определяется параметром remoteness. Все части эксперимента создаются на главной машине;
- 1 – разделение на равные вертикальные полосы. Их количество зависит от числа участвующих машин;
- 2 – разделение на вертикальные полосы, пропорциональные факторам производительности машин («throughput factor»)[1]. Их количество зависит от числа участвующих машин;
- другие значения могут быть приняты пользователем для обозначения разнообразных стратегий разделения при переопределении данной функции;

bool usethis – определяет, учитывать ли главную машину при разделении задачи. Если параметр равен true, то на ней тоже должен быть создан вспомогательный эксперимент;

int remoteness – определяет удаленность машин, которые следует привлекать к эксперименту (в настоящее время не поддерживается).

Функция может быть переопределена в потомке.

• public virtual bool ComputeCluster(bool gather=false) – функция посылает всем участникам эксперимента команды для осуществления итераций и ожидает ответов об их завершении. Параметр gather определяет, нужно ли собрать состояние всей решетки после данной итерации или нет. Функция может быть переопределена в потомке.

Во фрагменте кода, приводимом выше, при определении того, нужно ли собирать состояние решетки, использовалось выражение «Step+1-pCluster->Step>=p_iGather.Get()». Здесь значение переменной Step – номер шага в главном эксперименте. К нему прибавлена единица, так как внутри функции Compute его увеличение еще не произошло. Значение переменной pCluster->Step – номер шага, на котором осуществлялась последняя сборка состояния всей решетки. Если эти две величины расходятся не

[1] Величина фактора производительности определяется средой для каждой машины, на которой она запускается. Пользователь может самостоятельно определить его, выбрав в меню среды «Tools» | «Workstation Characteristics...».

меньше, чем на значение параметра p_iGather, то состояние всей решетки будет собрано.

Реализация для обобщенных координат. Компонент правил для клеточного автомата «Жизнь» в обобщенных координатах [4] даже проще, чем приведенный в подразд. «Простейшая реализация», так как задача становится одномерной. Функции Initialize и SubCompute примут следующий вид:

```
bool  CALifeRules::Initialize()
{
      if (!p_iAlive.IsAnalyzed()) return true;

      DATUM(CAGenBoolDatum);
     iq=0;

      for(CACell c=datum->z.a1; c<=datum->z.b1; c++)
    {
          if (datum->Get(c)) iq++;
    }
      p_iAlive.Set(iq);

      return true;
}

bool  CALifeRules::SubCompute(Zone& z)
{
      DATUM(CAGenBoolDatum);

      CACell neig[12]; // Массив для хранения всех соседей
      // Переменная, хранящая количество соседей
      unsigned short ncount=GET_METRICS->GetNeighboursCount();
      unsigned char alive; // Количество живых соседей текущей клетки
     iq=0;

      for(CACell c=z.a1; c<=z.b1; c++)
         {
            alive=0;

               // Получение всех соседей и размещение их в массиве
               pUnion->Metrics->GetNeighbours(c,neig);

               // Подсчет числа живых соседей текущей клетки
              for (int k=0; k<ncount; k++) {
                   if (datum->Get(neig[k])) alive++;
             }

               // Применение функции переходов и определение
               // значения анализируемого параметра
              if (datum->Get(c)) {
                     datum->Set(c,!p_bDie[alive]->Get());
                    if (!p_bDie[alive]->Get()) iq++;
             } else {
                     datum->Set(c,p_bBorn[alive]->Get());
                    if (p_bBorn[alive]->Get()) iq++;
             }
         }
       // Присваивание значения анализируемому параметру
       p_iAlive.Set(iq);

      return true;
}
```

В дополнение к этому необходимо изменить условия совместимости данного компонента, так как он требует обобщенной метрики. Таким образом, выражение его условий примет вид «Data.bool.*&Metrics.2D.generalized.*».

Для постановки эксперимента теперь придется использовать обобщенную метрику (например, стандартный компонент «Square Grids Generalized») и соответствующее хранилище данных (стандартный компонент «Booleans for Generalized»).

Необходимо отметить, что компонент, реализованный в подразд. «Простейшая реализация», тоже может работать с обобщенной метрикой, если указать условия совместимости «Data.bool.*&Metrics.2D.*». Этого не было сделано изначально только для ясности изложения.

Таким образом, описанный в настоящем подразделе компонент лишь оптимизирован под работу с одномерными координатами, однако принципиально новой функциональности он не реализует.

Решение уравнения теплопроводности

Теперь приведем пример решения физической задачи – уравнения теплопроводности с помощью пакета CAME&L. Как известно, уравнение имеет вид

$$\frac{\partial T}{\partial t} = a\Delta T, \qquad (1)$$

где $a = \frac{\lambda}{\rho c_p}$; λ – коэффициент теплопроводности материала; ρ – плотность материала; c_p – теплоемкость материала при постоянном давлении.

Каждая клетка автомата будет хранить значение температуры в соответствующей точке пространства, представляемое числом с плавающей точкой. Таким образом, автомат будет моделировать поле температур.

Для постановки эксперимента снова потребуется двумерная решетка из квадратов (хотя может использоваться и любая другая двумерная решетка), картезианская метрика. Помимо этого необходимо воспользоваться хранилищем вещественных данных для картезианских координат (стандартный компонент «Doubles for Cartesians»), а соответствующие правила требуется разработать.

Класс, реализующий эти правила, назовем CATCERules. Дабы не загромождать реализацию физическими величинами, будем считать параметр a из выражения (1), а также величины dx, dy и dt равными единице. В результате в качестве функции переходов будем использовать следующее выражение в конечных разностях:

$$T'(x,y) = \\ = \frac{T(x+1,y)+T(x-1,y)+T(x,y+1)+T(x,y-1)}{4}. \qquad (2)$$

В случае, если для хранилища данных будет включено хранение состояния на двух последовательных шагах, то можно считать, что в левой части выражения (2) стоит температура на следующем временном шаге, по сравнению с теми, что стоят в правой части. Если хранение на двух шагах не включено, то значения температуры в обеих частях вы-

ражения считаются для одного и того же временного шага, а тогда порядок обхода клеток при вычислении новых состояний приобретает значение, хотя и не принципиальное.

Предусмотрим в компоненте правил четыре анализируемых вещественных параметра: p_dAverageT – средняя температура на решетке, p_dMaxT – максимальная температура на решетке, p_dMinT – минимальная температура на решетке, p_dAverSelT – средняя температура среди выделенных пользователем в среде клеток.

Описание класса CATCERules будет иметь следующий вид:

```
class CATCERules:public CARules
{
public:
    // Конструктор и деструктор
    CATCERules();
    virtual ~CATCERules() {};

    // Определение имени, описания, иконки и условий
    // совместимости
    COMPONENT_NAME(TCE Solution (plain))
    COMPONENT_INFO(Thermal conductivity equation solution rules)
    COMPONENT_ICON(IDI_ICON)
    COMPONENT_REQUIRES(Data.double.*&Metrics.2D.cartesian.*)

    // Следующие функций класса CARules будут переопределены
    virtual bool Initialize();
    virtual bool SubCompute(Zone& z);

public:
    // Анализируемые параметры
    ParamDouble p_dAverageT;
    ParamDouble p_dMaxT;
    ParamDouble p_dMinT;
    ParamDouble p_dAverSelT;

public:
    // Карта анализируемых параметров
    APARAMETERS_COUNT(4)
    BEGIN_APARAMETERS
        PARAMETER(0,&p_dAverageT)
        PARAMETER(1,&p_dMaxT)
        PARAMETER(2,&p_dMinT)
        PARAMETER(3,&p_dAverSelT)
    END_APARAMETERS
};
```

Функции Initialize и SubCompute могут быть реализованы следующим образом:

```
bool CATCERules::Initialize()
{
    // Инициализацию следует производить, только если хотя бы один из
    // параметров анализируется
    if (!p_dAverageT.IsAnalyzed() && !p_dMaxT.IsAnalyzed()
        && !p_dMinT.IsAnalyzed() && !p_dAverSelT.IsAnalyzed())
        return true;

    DATUM(CATypedDatum<double>);
    CACell c; // Текущая клетка
    double val=0.0; // Значение из текущей клетки

    // Переменные для вычисления значений анализируемых параметров
    double avt=0.0,avselt=0.0;
    double maxt=datum->Get(GET_METRICS->ToCell(
        datum->z.a1,datum->z.a2,0));
    double mint=datum->Get(GET_METRICS->ToCell(
        datum->z.a1,datum->z.a2,0));

    for(CACell i=datum->z.a1; i<=datum->z.b1; i++)
        for(CACell j=datum->z.a2; j<=datum->z.b2; j++)
    {
            // Получение идентификатора текущей клетки
            // по декартовым координатам
            c=GET_METRICS->ToCell(i,j,0);
            val=datum->Get(c);

            avt+=val;
            if (val<mint) mint=val;
            if (val>maxt) maxt=val;
            if (p_dAverSelT.IsAnalyzed()) {
                if (find(GET_GRID->SelCells[1].begin(),
                    GET_GRID->SelCells.end(),c)!=
                    GET_GRID->SelCells.end()) avselt+=val;
        }
    }
    // Присваивание значений анализируемым параметрам
    p_dAverageT.Set(avt/(int)datum->z.GetVolume());
    p_dMaxT.Set(maxt);
    p_dMinT.Set(mint);
    if (GET_GRID->SelCells.size())
        p_dAverSelT.Set(avselt/GET_GRID->SelCells.size()); else
        p_dAverSelT.Set(0.0);

    return true;
}

bool CATCERules::SubCompute(Zone& z)
{
    DATUM(CATypedDatum<double>);
    CACell c; // Текущая клетка
    double val=0.0; // Значение из текущей клетки

    // Переменные для вычисления значений анализируемых параметров
    double avt=0.0,avselt=0.0;
    double maxt=datum->Get(GET_METRICS->ToCell(
        datum->z.a1,datum->z.a2,0));
    double mint=datum->Get(GET_METRICS->ToCell(
        datum->z.a1,datum->z.a2,0));

    for(CACell i=z.a1; i<=z.b1; i++)
        for(CACell j=z.a2; j<=z.b2; j++)
    {
            // Получение идентификатора текущей клетки
            // по декартовым координатам
            c=GET_METRICS->ToCell(i,j,0);

            // Вычисление нового состояния клетки по формуле (2)
            val= datum->Get(GET_METRICS->ToCell(i-1,j,0));
            val+=datum->Get(GET_METRICS->ToCell(i+1,j,0));
            val+=datum->Get(GET_METRICS->ToCell(i,j-1,0));
            val+=datum->Get(GET_METRICS->ToCell(i,j+1,0));
            val/=4.0;
            datum->Set(c,val);

            avt+=val;
            if (val<mint) mint=val;
            if (val>maxt) maxt=val;
            if (p_dAverSelT.IsAnalyzed()) {
                if (find(GET_GRID->SelCells.begin(),
                    GET_GRID->SelCells.end(),c)!=
                    GET_GRID->SelCells.end()) avselt+=val;
        }
    }
    // Присваивание значений анализируемым параметрам
```

[1] Переменная SelCells класса CAGrid хранит список клеток, выделенных пользователем. Таким образом, данное условие, накладываемое на результат, возвращаемый функцией find, позволяет определить, принадлежит ли клетка c ко множеству выделенных или нет.

```
    p_dAverageT.Set(avt/(int)z.GetVolume());
    p_dMaxT.Set(maxt);
    p_dMinT.Set(mint);
    if (GET_GRID->SelCells.size())
        p_dAverSelT.Set(avselt/GET_GRID->SelCells.size()); else
        p_dAverSelT.Set(0.0);

    return true;
}
```

Как видно из приведенного фрагмента кода, реализация функции `Initialize` отличается от `SubCompute` только отсутствием в ней вычисления нового значения клетки и наличием проверки, анализируется ли хотя бы один из параметров.

Здесь приведено простейшее решение уравнения теплопроводности. Однако, по аналогии с игрой «Жизнь», легко разработать варианты с зональной оптимизацией для многопроцессорной и кластерной вычислительных систем или обобщенных координат. Кроме того, нетрудно ввести параметры, определяющие физические характеристики системы, упомянутые в выражении (1).

Заключение

Примеры решения шести задач иллюстрирует тот факт, что библиотека CADLib предоставляет разработчику богатейший инструментарий. При работе с программным обеспечением CAME&L знание языка программирования C++ позволяет решать широкий спектр задач.

Последний тезис может показаться странным, так как владение языком C++ само по себе позволяет решать вычислительные задачи, не применяя специализированных пакетов. Кроме того, это – нетривиальный навык, рассчитывать на наличие которого у пользователя не правомерно.

Однако, во-первых, организация параллельных и распределенных вычислений с помощью любого из существующих средств требует от пользователя существенных программистских навыков. Во-вторых, пакет CAME&L берет на себя реализацию огромного количества вспомогательной функциональности – от сетевого взаимодействия до визуализации и сохранения экспериментов, что в полной мере оправдывает его использование.

В-третьих, программный интерфейс библиотеки CADLib построен из настолько общих соображений, что позволяет, например, разработать компонент правил, который в качестве параметра принимал бы описание клеточного автомата на неком специализированном языке, как FORTH [6, 7] или CARPET [8]. В результате, конечный пользователь будет избавлен от необходимости владеть языком C++.

Примеры, приведенные в разделе «Реализация игры "Жизнь"», наглядно показали, что функция `SubCompute` практически не меняется при переходе от одной вычислительной системы к другой. Таким образом, функциональность базового класса `CARules` может быть расширена так, что реализация правил для однопроцессорной, многопроцессорной и кластерной вычислительных систем будет осуществляться одним компонентом. Однако такой класс не входит в библиотеку CADLib, чтобы оставить пользователю большую свободу при реализации своих решений, схем разделения задачи и т. п.

Исходные коды всех обсуждаемых в настоящей работе компонентов доступны на сайте [9]. Там же выложен весь пакет CAME&L.

Работа выполнена при поддержке Российского фонда фундаментальных исследований по гранту № 05-07-90086 «Разработка среды и библиотеки "CAME&L" для организации параллельных и распределенных вычислений на основе клеточных автоматов».

Литература

1. **Gardner M.** The Fantastic Combinations of John Conway's New Solitaire Game «Life» // Scientific American. 1970. № 223.
2. **Гарднер М.** Математические досуги. М.: Мир, 1972.
3. **Гарднер М.** Крестики-нолики. М.: Мир, 1988.
4. **Naumov L.** Generalized Coordinates for Cellular Automata Grids // Computational Science – ICCS 2003. Part 2. Springer-Verlag. 2003.
5. **Гордеев А., Молчанов А.** Системное программное обеспечение. СПб.: Питер, 2001.
6. **Тоффоли Т., Марголус Н.** Машины клеточных автоматов. М.: Мир, 1991.
7. **Броуди Л.** Начальный курс программирования на языке FORTH. М.: Финансы и статистика, 1990.
8. **Spezzano G., Talia D.** Designing Parallel Models of Soil Contamination by the CARPET Language. 1998.
9. Сайт «CAMEL Laboratory» – http://camellab.spb.ru.