3.3.2 Простая реализация с использованием MPI.....92
3.3.3 Реализация с поддержкой вложенных сетей.....95

3.4 Бильярдные шары 103

3.5 Сумматор 112

Глава 4. Сети Петри.....121

4.1 Краткое введение в теорию сетей Петри 121
4.1.1 Знакомство с сетями Петри 121
4.1.2 Строго иерархические сети 125
4.1.3 Параллельные вычисления и синхронизация 127
4.1.4 Задача об обедающих философах 130

4.2 Пример реализации механизма сетей Петри 133
4.2.1 Функционирование строго иерархических сетей.....134
4.2.2 Выполнение параллельных процессов 145

Заключение.....157

Приложение 1. Шаблоны классов матрицы и вектора.....159

Приложение 2. Классы построения и выполнения комплекса работ.....163

Приложение 3. Классы построения и выполнения сетей конечных автоматов 167

Приложение 4. Классы построения и выполнения сетей Петри.....174

Библиографический список 182

Оглавление.....186

Part 2

Глава 2. Ярусно-параллельная форма программы

В этой главе мы расскажем об одном из самых естественных и просто реализуемых подходов к распараллеливанию некоторой комплексной задачи. Будем исходить из предположения, что решаемая задача обладает параллелизмом независимых ветвей, т.е. она состоит из некоторого количества подзадач, процесс выполнения каждой из которых не связан с другими. При этом каждая задача, возможно, по своим входным данным находится в зависимости от результатов выполнения других подзадач.

Способы решения задач, подобных описанной, относятся к методам сетевого планирования и управления [5, 12]. В соответствии с принятой в этой области терминологией, будем называть в дальнейшем подзадачи – работами, а всю распараллеливаемую задачу – комплексом работ.

2.1 Цель и механизм построения ЯПФ

Графическое представление схемы зависимостей работ друг от друга называется сетевым графиком работ. Сетевой график работ представляется направленным ациклическим графом, т.е. никакая работа не должна прямо либо косвенно зависеть от самой себя. К примеру, на рис. 7 можно увидеть сетевой график некоторой комплексной работы, состоящей из восьми работ. Прямоугольниками здесь обозначены работы, стрелками – зависимости между ними по входным и выходным данным.

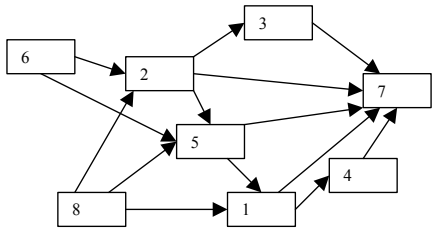

Рис. 7

В сетевом планировании используются два способа представления сетевого графика работ. В первом случае, как на рис. 7, работы представляются вершинами графа, зависимости – дугами.

Оглавление

Введение ... *3*

О целях издания ... **3**

О проблеме параллельного программирования ... **5**

Об используемой терминологии ... **8**

Глава 1. Интерфейсы и технологии параллельного программирования ... *10*

1.1 Интерфейс OpenMP ... **10**

1.1.1 Первая программа – ряд Лейбница ... 11
1.1.2 Основные конструкции параллельного выполнения ... 17
1.1.3 Некоторые вспомогательные директивы ... 22
1.1.4 Разделение данных ... 24
1.1.5 Runtime-функции ... 26
1.1.6 Вычисление определенного интеграла ... 30

1.2 Интерфейс передачи сообщений MPI ... **32**

1.2.1 Краткое описание предоставляемых функций ... 33
1.2.2 Снова ряд Лейбница ... 36
1.2.3 Неравномерное распределение вычислений ... 38
1.2.4 Умножение матрицы на вектор ... 45
1.2.5 Перемножение матриц ... 51

Глава 2. Ярусно-параллельная форма программы ... *60*

2.1 Цель и механизм построения ЯПФ ... **60**

2.2 Реализация выполнения комплекса работ при использовании фиксированного количества параллельных ресурсов ... **65**

2.3 Случай неравномерной длительности работ ... **69**

Глава 3. Сети конечных автоматов ... *75*

3.1 Программирование конечных автоматов ... **75**

3.2 Параллелизм сетей конечных автоматов ... **80**

3.3 Пример программной реализации ... **83**

3.3.1 Реализация с использованием OpenMP ... 86

```
    delete dynamic_cast<calculator_type *>(pfsm);
  }
 };

public:

 summator_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Такой набор классов может быть использован следующим кодом:

```
// инициализация вектора входных значений сети
vector<fsm_type::data_type> vin;
// ...

// количество автоматов - контроллер+вычислители
int fsanum = 1 + DEF_CALCNUM;

// фабрика сети и сеть
summator_type::factory_type
 factory(vin.size(), summator_type::OUTPUT_NUMBER, fsanum);
summator_type fsmnet(factory);

// передать входные данные
fsmnet.put_input(vin);

// вычисленное значение суммы
fsm_type::data_type sum = 0;
do
{
 // выполнить такт работы сети
 fsmnet.do_work();

 // получить очередные выходные данные
 vector<fsm_type::data_type> vout = fsmnet.get_output();

 // если результат есть, сохраним его
 if (vout[summator_type::OUTPUT_ON])
  sum = vout[summator_type::OUTPUT_RESULT];

} while (!fsmnet.is_off());
```

Количество входов сети определяется на основе длины вектора входных значений. После передачи входной последовательности в сеть выполняется полный цикл работы сети. В течение цикла осуществляется проверка наличия выходного результата и его сохранение.

Во втором случае работы представляются дугами, вершинами представляются события их завершения. Общие правила построения таких сетевых графиков приведены в [12].

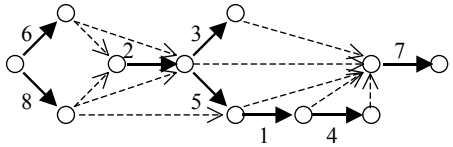

Рис. 8

Пример представления вторым способом комплекса работ, изображенного на рис. 7, приведен на рис. 8. Здесь пунктирными стрелками обозначены так называемые фиктивные работы, не занимающие ресурсов и характеризующие логическую связь. Разумеется, на этом графике можно опустить фиктивные работы, характеризующие связи, которые неявно вытекают из других, в результате чего получаем заключительное представление сетевого графика работ (рис. 9).

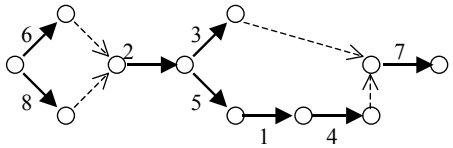

Рис. 9

Оба способа имеют свои достоинства в тех или иных условиях. Второй способ удобен для анализа временных характеристик работ и не обременен лишними связями, поэтому широко используется в сетевом планировании. Первый способ не нуждается в использовании фиктивных работ и обеспечивает более высокую гибкость при добавлении неучтенных зависимостей. Поскольку мы рассматриваем случай, когда анализ временных характеристик не актуален, мы будем пользоваться первым способом представления сетевого графика.

В качестве альтернативы графическому представлению комплекс работ может быть представлен структурной таблицей комплекса работ, в которой указываются работы, включенные в

комплекс, а также зависимости каждой работы от других (рис. 10).

Job	Dependence
1	5, 8
2	6, 8
3	2
4	1
5	2, 6, 8
6	-
7	1, 2, 3, 4, 5
8	-

Рис. 10

Решение как таковых задач сетевого планирования и управления сводится к определению оптимального распределения ресурсов с целью получения минимального времени выполнения всего комплекса на основе известных зависимостей времени выполнения отдельных работ от количества выделенных им ресурсов.

При выполнении параллельных вычислений в такой постановке рассматривать задачу оказывается затруднительно и зачастую бессмысленно, поскольку в однородной системе оказывается нечем управлять – мы не можем повлиять на скорость выполнения конкретной подзадачи, а в неоднородной оказывается сложно задать зависимости величин времени выполнения каждой подзадачи.

Нам интересен лишь первый этап решения задачи поиска критического пути – упорядочение заданного сетевого графика работ и построения ярусно-параллельной формы программы [1, 5, 12].

Процедура построения ярусно-параллельной формы (далее – ЯПФ) достаточно проста и заключается в представлении сетевого графика в форме последовательных ярусов работ, в пределах каждого из которых работы могут выполняться независимо друг от друга. Построение ЯПФ легко осуществляется на основе структурной таблицы комплекса работ и сводится к добавлению столбца с присвоенными номерами ярусов. Работами нулевого яруса считаются все работы, выполнение которых не зависит от


```
{}

int number_fsm(void) const { return m_fsmnum; }

int number_input(void) const { return m_inputnum; }

int number_output(void) const { return m_outputnum; }

linkstore_type get_links(void) const
{
 linkstore_type store;

 // входы сети прицепим к контроллеру
 for (int i = 0; i < m_inputnum; i++)
  store.add(PSEUDOFSM_NETINPUT, i, FSM_CONTROL, i);

 // выходы сети - первые два выхода от контроллера
 for (int i = 0; i < m_outputnum; i++)
  store.add(FSM_CONTROL, i, PSEUDOFSM_NETOUTPUT, i);

 // все вычислители к контроллеру подцепим
 for (int i = 0; i < m_fsmnum - FSM_CALCULATOR_0; i++)
 {
  // аргументы на вычислители с контроллера (после выходов сети)
  for (int j = 0; j < calculator_type::INPUT_NUMBER; j++)
   store.add(
    FSM_CONTROL,
    m_outputnum + calculator_type::INPUT_NUMBER * i + j,
    FSM_CALCULATOR_0 + i, j);

  // выход вычислителя - к контроллеру, после входов сети
  store.add(
   FSM_CALCULATOR_0 + i, calculator_type::OUTPUT_RESULT,
   FSM_CONTROL, m_inputnum + i);
 };
 return store;
}

fsm_abstract_type *create_fsm(int i)
{
 fsm_abstract_type *pfsm;
 if (i == FSM_CONTROL)
  pfsm = new control_type(m_inputnum, m_fsmnum - 1);
 else
  pfsm = new calculator_type();
 return pfsm;
}

void destroy_fsm(int i, fsm_abstract_type *pfsm)
{
 if (i == FSM_CONTROL)
  delete dynamic_cast<control_type *>(pfsm);
 else
```

```
  add_handler(STATE_OUT, handler_type(&control_type::do_off));
 }
};
```

Обработчик выключенного состояния do_start проверяет наличие на входе контроллера последовательности ненулевых чисел, после чего, в случае наличия, помещает их в очередь m_queue и распределяет по вычислителям с помощью функции dispatch. Функция dispatch возвращает количество задействованных вычислителей, на основе которого задается следующее состояние.

Для всех состояний от «0» до «N» определен обработчик do_calc. В начале функции do_calc осуществляется сбор результатов вычислений предыдущего такта со всех вычислителей с помощью функции collect. Далее выполняется проверка факта завершения вычислений, который определяется наличием в очереди лишь одного элемента и отсутствием активных вычислителей. В случае если хотя бы одно из этих условий не выполняется, производится попытка следующего распределения. Если же вычисления закончены, происходит переход к состоянию «Вывод результата». Его обработчик do_off просто переводит автомат в выключенное состояние.

Автоматная сеть представляется классом summator_type, который объявляет константы для обозначения автоматов и выходов сети, а также класс фабрики сети, создающий автоматы и формирующий связи между ними в соответствии с рис. 22:

```
class summator_type: public fsmnet_type
{
public:

 enum {FSM_CONTROL, FSM_CALCULATOR_0};
 enum {OUTPUT_RESULT, OUTPUT_ON, OUTPUT_NUMBER};

 class factory_type: public factory_abstract_type
 {
 private:

  int m_inputnum, m_outputnum, m_fsmnum;

 public:

  factory_type(int inputnum, int outputnum, int fsmnum):
   m_inputnum(inputnum), m_outputnum(outputnum), m_fsmnum(fsmnum)
```


других. Работы первого яруса – те, которые опираются только на работы нулевого яруса. Работы каждого яруса из последующих должны опираться только на работы более ранних по отношению к нему ярусов, среди которых должна быть как минимум одна работа непосредственно предшествующего яруса. На рис. 11 можно увидеть пример распределения по ярусам работ комплекса, изображенного на рис. 7.

Job	Dependence	Level
1	5, 8	3
2	6, 8	1
3	2	2
4	1	4
5	2, 6, 8	2
6	-	0
7	1, 2, 3, 4, 5	5
8	-	0

Рис. 11

После выполнения распределения по ярусам всех работ комплекса, сетевой график может быть изображен с учетом разбиения по ярусам. Пример разбитого по ярусам комплекса работ можно увидеть на рис. 12.

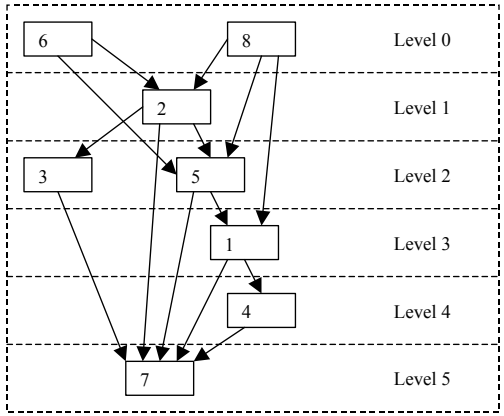

Рис. 12

Такая форма организации программы носит название ЯПФ программы. Преимущество приведения программы к ЯПФ за-

ключается в возможности параллельного выполнения в рамках одного яруса всех включенных в него работ. Количество работ в каждом ярусе называется шириной яруса, количество ярусов – высотой ЯПФ, максимальная ширина яруса – шириной ЯПФ программы.

Эти характеристики при некоторых допущениях позволяют нам судить о свойствах построенной таким образом программы. В частности, при условии, что все работы приблизительно одинаковы по длительности, ширина ЯПФ характеризует необходимое максимальное количество требуемых параллельных ресурсов во время выполнения. В случае наличия необходимого количества параллельных ресурсов время выполнения комплексной работы будет приблизительно равно произведению времени выполнения одной работы на высоту ЯПФ.

В [1] говорится, что распараллеливание программ с использованием ЯПФ плохо согласовано с конструкциями существующих языков программирования, чем обусловлена сложность его выполнения и соответствующее крайне редкое использование. Действительно, большинство популярных сегодня языков программирования, как правило, изначально предназначены для последовательных вычислений и поэтому плохо приспособлены для любого распараллеливания. Однако следует учитывать, что написание сложной программы (а параллельная программа, как правило, является сложной) всегда требует усилий, и поиск легких путей в данном случае работает в большинстве случаев против нас. В связи с этим для построения параллельных программ мы предложим далеко не новый подход, заключающийся в отделении функциональности используемой схемы распараллеливания, а именно построения ЯПФ и выполнения как такового распараллеливания, от специфики конкретной задачи. В этом случае написание кода, отвечающего за параллельное выполнение комплекса работ, разумеется, будет не легче, чем прежде, однако может быть выполнено лишь один раз, при этом использоваться такой код может при решении широкого круга задач.


```
    m_output[OUTPUT_RESULT] = 0;
    m_output[OUTPUT_ON] = 0;
   }
   else
   {
    // иначе продолжим быть выключенными
    state = STATE_OFF;
    fill(m_output.begin(), m_output.end(), 0);
   };
   return state;
  }

  state_type do_calc(state_type state)
  {
   // сложить все ненулевые предыдущие результаты в конец очереди
   collect();
   // выход, если активно 0 вычислителей и в очереди один элемент
   if (state == STATE_CALC_NUMBER_0 && m_queue.size() == 1)
   {
    state = STATE_OUT;
    m_output[OUTPUT_RESULT] = m_queue.front();
    m_output[OUTPUT_ON] = 1;
   }
   else
   {
    // распределить пары из начала очереди
    state = STATE_CALC_NUMBER_0 + dispatch();
    m_output[OUTPUT_RESULT] = 0;
    m_output[OUTPUT_ON] = 0;
   };
   return state;
  }

  state_type do_off(state_type state)
  {
   // перейдем в выключенное состояние
   state = STATE_OFF;
   fill(m_output.begin(), m_output.end(), 0);
   return state;
  }

 public:
  control_type(int argnum, int calcnum):
   fsm_type(argnum + calcnum,
    calcnum * calculator_type::INPUT_NUMBER + 2),
   m_argnum(argnum), m_calcnum(calcnum)
  {
   add_handler(STATE_OFF, handler_type(&control_type::do_start));

   for (int i = 0; i <= calcnum; i++)
    add_handler(
     STATE_CALC_NUMBER_0 + i, handler_type(&control_type::do_calc));
```

```
private:
 deque<data_type> m_queue;
 int m_argnum; // количество аргументов сумматора
 int m_calcnum; // количество вычислителей

 // собрать со входов все ненулевые результаты вычислений
 void collect(void)
 {
  for (int i = 0; i < m_calcnum; i++)
  {
   data_type result = m_input[m_argnum + i];
   if (result)
    m_queue.push_back(result);
  };
 }

 // распределить пары данных, сколько есть в наличии, на вычислители
 int dispatch(void)
 {
  int num =
   (m_queue.size() >= (m_calcnum << 1)) ?
    m_calcnum : (m_queue.size() >> 1);
  for (int i = 0; i < num; i++)
  {
   m_output[OUTPUT_ARG1_0 + calculator_type::INPUT_NUMBER * i] =
    m_queue.front();
   m_queue.pop_front();
   m_output[OUTPUT_ARG2_0 + calculator_type::INPUT_NUMBER * i] =
    m_queue.front();
   m_queue.pop_front();
   m_output[OUTPUT_CALC_ON_0 + calculator_type::INPUT_NUMBER * i]=1;
  };
  for (int i = num; i < m_calcnum; i++)
  {
   m_output[OUTPUT_ARG1_0 + calculator_type::INPUT_NUMBER * i] = 0;
   m_output[OUTPUT_ARG2_0 + calculator_type::INPUT_NUMBER * i] = 0;
   m_output[OUTPUT_CALC_ON_0 + calculator_type::INPUT_NUMBER * i]=0;
  };
  return num;
 }

 state_type do_start(state_type state)
 {
  // проверим наличие сигналов на входе (поищем ненулевой)
  bool noinput = (m_input.end() ==
   find_if(m_input.begin(), m_input.end(),
    not1(logical_not<data_type>())));
  // если есть, переключимся на обработку
  if (!noinput)
  {
   // сложим все входные данные в очередь
   m_queue.assign(m_input.begin(), m_input.begin() + m_argnum);
   state = STATE_CALC_NUMBER_0 + dispatch();
```


2.2 Реализация выполнения комплекса работ при использовании фиксированного количества параллельных ресурсов

Прежде чем переходить к описанию предлагаемых вариантов реализации, следует оговорить следующий момент. В [1] было указано, что вследствие того, что любая работа текущего яруса находится в зависимости от хотя бы одной работы предыдущего яруса, она не может быть начата раньше, чем завершится выполнение работ предыдущего яруса.

Следует отметить, что это не совсем верно, однако рассмотрение с такой точки зрения помимо своих минусов имеет и свои плюсы. При подобной реализации нам в любой момент известно необходимое количество параллельных ресурсов, требуемых для выполнения работ, поскольку оно равно ширине выполняющегося в текущий момент яруса.

Следует также отметить, что такой подход может быть достаточно эффективным и простым для реализации при условии, что в пределах каждого яруса все работы приблизительно одинаковы по длительности.

В противном же случае, когда длительности существенно различаются, а зависимостей между работами не очень много, могут возникнуть существенные потери времени на ожидание некоторой работой завершения какой-либо работы предыдущего яруса, от которой она не зависит.

Поскольку в реальности встречаются разные ситуации, мы рассмотрим оба варианта. Вначале рассмотрим реализацию более простого случая, когда все работы выполняются строго поярусно, т.е. никакая работа следующего яруса не начинается до тех пор, пока не закончат выполнение все работы текущего яруса.

Для простоты и наглядности абстрагируемся от необходимости передачи входных и выходных данных между работами и сконцентрируемся на распределении их выполнения между параллельными ресурсами. Будем считать, что данные передаются через разделяемые ресурсы внутри работ.

В приложении 2 приведен код предлагаемых классов, распараллеленный средствами интерфейса OpenMP. Среди них, прежде всего, абстрактный класс работы:

```
// абстрактный интерфейс работы
class job_abstract_type
{
public:
 // выполнение работы
 // получение исходных данных и вывод результата
 // выполняются внутри через разделяемые ресурсы
 virtual void run(void) = 0;
};
```

Все классы работ в вызывающем коде должны быть унаследованы от этого класса и должны переопределять функцию run, внутри которой должно осуществляться выполнение работы.

Другой класс – класс комплекса работ jobcomplex_type. Поскольку комплекс работ, в сущности, также является работой, класс jobcomplex_type унаследован от абстрактного класса job_abstract_type и, в принципе, может быть использован в качестве работы при формировании комплекса работ более высокого уровня сложности.

В классе jobcomplex_type объявляются необходимые структуры данных для заполнения комплекса работами и зависимостями. Для удобства добавления зависимостей указатели на работы хранятся вместе с отображением на их номера. Номера работам присваиваются в соответствии с порядком добавления. Для добавления работ и зависимостей вызывающему коду предоставляются функции add_job и add_dependence соответственно.

Реализация функции выполнения комплекса работ run начинается с формирования списка работ и таблицы зависимостей на основе заполненных ранее структур данных с помощью функций get_joblist и get_deplist. После этого вызывается функция build, которая осуществляет построение ЯПФ на основе таблицы зависимостей работ. После осуществления построения, в цикле по номерам ярусов выполняются все работы. Работы каждого яруса выполняются параллельно, для чего используется директива OpenMP, при этом вследствие наличия неявного барьера до момента завершения всех работ текущего яруса следующий не начинается.

Функция build, осуществляющая построение ЯПФ, начинается с определения ярусов всех работ. С этой целью определяется множество работ, для которых ярусы еще не определены, изна-

один элемент, состояние меняется на «Вывод результата». На выходе при этом единственный элемент очереди и сигнал о выводе результата. На следующем такте выключается контроллер и вся сеть.

Класс вычислителя calculator_type определяет один обработчик на оба состояния:

```
class calculator_type: public fsm_type
{
public:
 enum {INPUT_ARG1, INPUT_ARG2, INPUT_ON, INPUT_NUMBER};
 enum {OUTPUT_RESULT, OUTPUT_NUMBER};
 enum {STATE_OUT};

private:
 state_type do_calc(state_type state)
 {
  if (m_input[INPUT_ON])
  {
   state = STATE_OUT;
   m_output[OUTPUT_RESULT]= m_input[INPUT_ARG1]+m_input[INPUT_ARG2];
  }
  else
  {
   state = STATE_OFF;
   m_output[OUTPUT_RESULT] = 0;
  };
  return state;
 }
public:
 calculator_type():
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&calculator_type::do_calc));
  add_handler(STATE_OUT, handler_type(&calculator_type::do_calc));
 }
};
```

В случае наличия входного сигнала включения, вычислитель производит суммирование и вывод результата. В противном случае переходит в выключенное состояние.

Класс контроллера control_type определяет три обработчика состояний:

```
class control_type: public fsm_type
{
public:
 enum {OUTPUT_RESULT, OUTPUT_ON, OUTPUT_ARG1_0, OUTPUT_ARG2_0,
OUTPUT_CALC_ON_0};
 enum {STATE_OUT, STATE_CALC_NUMBER_0};
```

литель очередную пару значений и управляющий сигнал, а также выдает на выход сети результат суммирования всей последовательности.

Принцип функционирования сети заключается в следующем. Сеть начинает работу при получении на входе последовательности с ненулевыми элементами. Все элементы попадают в некоторую очередь, хранимую в контроллере и распределяемую попарно контроллером на вычислители. Результаты суммирования с выходов вычислителей также помещаются контроллером в конец очереди.

Вместе с парой чисел на каждый вычислитель подается управляющий сигнал «вкл». Когда длина очереди становится меньше удвоенного количества вычислителей, те вычислители, на которые не хватило работы, получают управляющий сигнал «выкл».

На первом такте работы сети полезную работу выполняет только контроллер – принимает входную последовательность и передает пары на входы вычислителей. На втором такте вычислители производят суммирование, контроллер же готовит следующие пары. На этом такте выходы с вычислителей не учитываются, поскольку после первого такта они были нулевыми. На третьем такте работы контроллера результаты вычислений второго такта добавляются в конец очереди, и готовится третье распределение пар, пока вычислители обрабатывают второе.

Каждый вычислитель пребывает либо в выключенном состоянии, либо в состоянии выдачи результата. В последнее состояние вычислитель переходит при получении управляющего сигнала «вкл» после выполнения суммирования двух переданных на вход аргументов.

Состояние контроллера характеризует количество работающих в текущий момент вычислителей, т.е. количество распределенных в последний раз пар чисел. В начале работы обычное состояние контроллера – «N». Когда процесс близится к завершению вычислений, состояние меняется на меньшее «N», в соответствии с текущим количеством распределенных пар. По достижении состояния «0», если в очереди контроллера остался лишь

чально содержащее номера всех работ. Далее до момента опустошения этого множества выполняется цикл, в теле которого на основе таблицы зависимостей производится попытка определить ярусы каких-либо работ из оставшихся, после чего выполняется исключение найденных работ из множества неопределенных и переход к определению работ следующего яруса.

При попытке определить номер яруса работы осуществляется проход по всем ее зависимостям. Если какая-либо работа, от которой текущая зависит, все еще в списке неопределенных, то и текущая работа остается неопределенной. В противном случае среди всех зависимостей определяется максимальный номер яруса, после чего полученное значение с увеличением на единицу присваивается номеру яруса текущей работы.

После определения номеров ярусов всех работ на основе них строится как таковая ЯПФ, представляемая в программе в виде списка списков, каждый список в котором характеризует ярус работ, а элементы – их номера.

Использование приведенных классов оказывается довольно простым и наглядным. К примеру, фрагмент программы, реализующий выполнение комплекса работ, приведенного на рис. 7, может выглядеть следующим образом:

```
// создание объектов работ
// j1, j2, j3, j4, j5, j6, j7, j8
// ...

// создание и наполнение комплекса работ
jobcomplex_type jobcomplex;
// добавление списка работ
jobcomplex.add_job(j1);
jobcomplex.add_job(j5);
jobcomplex.add_job(j3);
jobcomplex.add_job(j2);
jobcomplex.add_job(j7);
jobcomplex.add_job(j4);
jobcomplex.add_job(j6);
jobcomplex.add_job(j8);
// добавление списков зависимостей
jobcomplex.add_dependence(j1, j5);
jobcomplex.add_dependence(j1, j8);
jobcomplex.add_dependence(j2, j6);
jobcomplex.add_dependence(j2, j8);
jobcomplex.add_dependence(j3, j2);
jobcomplex.add_dependence(j4, j1);
```

```
jobcomplex.add_dependence(j5, j2);
jobcomplex.add_dependence(j5, j6);
jobcomplex.add_dependence(j5, j8);
jobcomplex.add_dependence(j7, j1);
jobcomplex.add_dependence(j7, j2);
jobcomplex.add_dependence(j7, j3);
jobcomplex.add_dependence(j7, j4);
jobcomplex.add_dependence(j7, j5);

// выполнение комплекса работ
jobcomplex.run();
```

Каждая работа представляется объектом класса, унаследованного от job_abstract_type. После их создания производится создание и заполнение объекта комплекса работ, а именно добавление работ и зависимостей между ними в соответствии со структурной таблицей комплекса. Наконец, производится выполнение комплекса работ.

Рассмотрим теперь вариант модификации приведенного в приложении 2 кода для распределенного выполнения с использованием интерфейса MPI. В сущности, изменениям подверглась лишь функция run:

```
// выполнение всего комплекса работ
void run(void)
{
 // получим список подлежащих выполнению работ
 joblist_type joblist = get_joblist();
 // построим ярусно-параллельную структуру номеров работ
 multilevel_type multilevel = build(get_deplist());

 int rank, size;
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 // выполним по очереди каждый ярус работ
 for (int l = 0; l < multilevel.size(); l++)
 {
  MPI_Barrier(MPI_COMM_WORLD);
  // распределим каждый ярус работ циклически
  for (int i = rank; i < multilevel[l].size(); i += size)
   joblist[multilevel[l][i]]->run();
 };
}
```

Для распределенного выполнения работ потребовалось внести явную барьерную синхронизацию между ярусами и циклически распределить выполнение работ каждого яруса по процессам коммуникатора MPI_COMM_WORLD.

программных автоматных сетей.

При описании параллельных алгоритмов часто приводится алгоритм сдваивания – алгоритм параллельного суммирования последовательности чисел некоторой длины n. Суть его заключается в том, что вся последовательность разбивается по парам, после чего все пары суммируются параллельно. Результаты снова разбиваются попарно и суммируются, и т.д. Для любого n сумма всей последовательности будет получена за количество шагов, равное $\lceil \log_2 n \rceil$, если алгоритм работает в условиях неограниченного параллелизма.

При реализации алгоритма суммирования сдваиванием в виде автоматной сети мы будем учитывать реальное отсутствие неограниченного параллелизма и возложим выполнение операции суммирования на конечный набор вычислителей.

Реализуем автоматную сеть в виде, представленном на рис. 22. Сеть содержит N автоматов, выполняющих непосредственно вычисление суммы двух чисел, и один автомат, управляющий вычислением суммы всей последовательности.

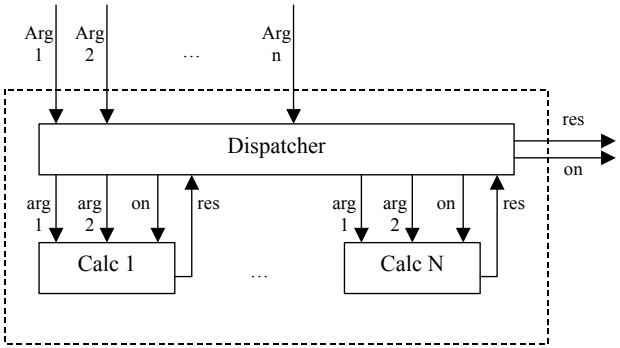

Рис. 22

Каждый вычисляющий автомат (далее – вычислитель) получает на входе пару чисел и управляющий сигнал, на выходе выдает результат суммирования. Управляющий автомат (далее – контроллер) в качестве входных сигналов получает всю последовательность суммируемых чисел, а также результаты суммирования с вычислителей. В качестве выхода подает на каждый вычис-

```
    delete dynamic_cast<table_type *>(pfsm);
   else
    delete dynamic_cast<ball_type *>(pfsm);
  }
 };

public:
 billiard_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Как говорилось выше, входами сети являются количество циклов выполнения и инициализационные данные для каждого шара. Функция get_links класса фабрики сети задает связи между автоматами, входами и выходами сети в соответствии с рис. 21.

Выполнение цикла работы такой сети может осуществлять следующий фрагмент кода:

```
billiard_type::factory_type factory(5, 6);
billiard_type fsmnet(factory);

vector<fsm_type::data_type> vin(billiard_type::INPUT_NUMBER);
vin[billiard_type::INPUT_CYCLES] = 20;
vin[billiard_type::INPUT_INIT_X1] = 1;
vin[billiard_type::INPUT_INIT_Y1] = 1;
vin[billiard_type::INPUT_INIT_DIR1] = ball_type::STATE_DOWN_RIGHT;
vin[billiard_type::INPUT_INIT_X2] = 4;
vin[billiard_type::INPUT_INIT_Y2] = 4;
vin[billiard_type::INPUT_INIT_DIR2] = ball_type::STATE_UP_LEFT;

fsmnet.put_input(vin);
do
 fsmnet.do_work();
while (!fsmnet.is_off());
```

После создания объектов фабрики и сети формируется входной вектор сети. Далее производится передача входного вектора непосредственно в сеть и выполняется цикл работы сети. Автоматная сеть устроена так, что входной вектор может не меняться на протяжении работы сети, выходных же значений нет вообще, вследствие чего чтение и запись входных и выходных значений внутри цикла выполнения не производится.

3.5 Сумматор

Теперь рассмотрим другой пример, в отличие от предыдущего, мало привязанный к реальности, но в некоторой степени также подходящий для иллюстрации возможностей построения

Для использования модифицированного таким образом набора классов приведенный выше фрагмент вызывающей программы должен быть дополнен лишь вызовами MPI_Init и MPI_Finalize.

2.3 Случай неравномерной длительности работ

Теперь рассмотрим случай, когда длительности работ существенно отличаются друг от друга. Очевидно, что если в такой ситуации в конце каждого яруса выполнять барьерную синхронизацию между всеми параллельными ресурсами, будут возникать существенные задержки общего времени выполнения вследствие простоя тех ресурсов, которые закончили выполнение работы раньше. В случае если при этом некоторая работа текущего яруса зависит лишь от тех работ предыдущего, которые уже завершились, она может быть уже начата и, возможно даже, завершена, однако она вынуждена ожидать достижения барьера всеми остальными.

В этой связи снимем для запуска работы условие необходимости завершения всех работ предыдущего яруса. Вместо этого вернемся к изначальному условию необходимости завершения тех работ, от которых данная работа непосредственно зависит. В этой ситуации могут сократиться задержки по времени, однако это не всегда будет «бесплатно». Зачастую в таких условиях требуемое количество параллельных ресурсов может превышать ширину ЯПФ.

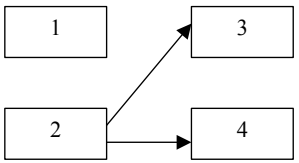

Рис. 13

Поясним это на примере. На рис. 13 приведен пример комплекса работ, ширина ЯПФ которого равна двум. В случае если вторая работа окажется гораздо короче первой, для наискорейше-

го выполнения комплекса работ потребуются три параллельных ресурса – для одновременного выполнения первой, третьей и четвертой работ. На рис. 14 показано, как такой комплекс может выполняться во времени строго по ярусам с барьером между ними (слева) и при уплотнении с учетом индивидуальных зависимостей (справа). В таких условиях количество требуемых параллельных ресурсов в общем случае может очень существенно отличаться от ширины ЯПФ, вплоть до того, что в некоторых случаях может приближаться к общему количеству работ.

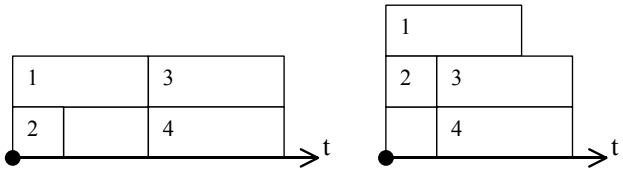

Рис. 14

В связи с неопределенностью необходимого количества параллельных ресурсов, реализация такого подхода с использованием MPI оказывается затруднительной вследствие статического существования группы процессов. Помимо этого, для подобной реализации необходимы достаточно низкоуровневые механизмы синхронизации, вследствие чего интерфейс OpenMP также оказывается неподходящим.

В этом случае может подойти какой-либо более низкоуровневый программный интерфейс, к примеру, для многопоточной реализации могут быть использованы интерфейсы Win32 API или POSIX Threads (pthreads). Мы приведем вариант многопоточной реализации с использованием интерфейса Win32 API.

Прежде всего, оговорим функциональность предлагаемого кода. Каждая работа будет выполняться в своем потоке. При этом вместо ожидания завершения каждого яруса работ перед началом следующего, внесем перед выполнением каждой работы ожидание завершения тех работ, от которых она зависит. Такие ожидания вынесем в начало каждого созданного для выполнения работы потока и поместим непосредственно перед


```
    store.add(
     PSEUDOFSM_NETINPUT, INPUT_INIT_X1 + i,
     FSM_BALL1, ball_type::INPUT_INIT_X + i);
   };
   // входные координаты и направление второго шара
   for (int i = 0; i <= INPUT_INIT_DIR2 - INPUT_INIT_X2; i++)
   {
    store.add(
     PSEUDOFSM_NETINPUT, INPUT_INIT_X2 + i,
     FSM_BALL2, ball_type::INPUT_INIT_X + i);
   };
   // выходы стола - его границы
   for (int i = 0; i < table_type::OUTPUT_NUMBER; i++)
   {
    store.add(
     FSM_TABLE, i,
     FSM_BALL1, ball_type::INPUT_BOUND_LEFT + i);
    store.add(
     FSM_TABLE, i,
     FSM_BALL2, ball_type::INPUT_BOUND_LEFT + i);
   };
   // выходы шаров - на вход стола и друг другу
   for (int i = 0; i < ball_type::OUTPUT_NUMBER; i++)
   {
    // входы стола - координаты и направления шаров
    store.add(
     FSM_BALL1, i,
     FSM_TABLE, table_type::INPUT_X1 + i);
    store.add(
     FSM_BALL2, i,
     FSM_TABLE, table_type::INPUT_X2 + i);
    // оба шара связаны между собой
    store.add(
     FSM_BALL1, i,
     FSM_BALL2, ball_type::INPUT_OTHER_X + i);
    store.add(
     FSM_BALL2, i,
     FSM_BALL1, ball_type::INPUT_OTHER_X + i);
   };
   return store;
  }
  fsm_abstract_type *create_fsm(int i)
  {
   fsm_abstract_type *pfsm;
   if (i == FSM_TABLE)
    pfsm = new table_type(m_hsize, m_vsize);
   else
    pfsm = new ball_type();
   return pfsm;
  }
  void destroy_fsm(int i, fsm_abstract_type *pfsm)
  {
   if (i == FSM_TABLE)
```

включенном и выключенном. В конструкторе класса table_type обоим состояниям назначается обработчик do_out.

Внутри функции do_out выполняется отображение шаров и наращивание значения счетчика циклов, после чего на основе этого значения автомат может перейти в выключенное состояние или остаться во включенном. В конце функции выполняется вывод границ стола. В случае выключения все границы обнуляются, в связи с чем на следующем такте завершат работу шары.

Наконец, все описанные автоматы должны быть объединены в сеть. Для этого реализуется класс billiard_type, описывающий входы и выходы сети, а также объявляющий класс фабрики сети:

```
class billiard_type: public fsmnet_type
{
public:
 enum {FSM_TABLE, FSM_BALL1, FSM_BALL2, FSM_NUMBER};
 enum {
  INPUT_CYCLES,
  INPUT_INIT_X1,
  INPUT_INIT_Y1,
  INPUT_INIT_DIR1,
  INPUT_INIT_X2,
  INPUT_INIT_Y2,
  INPUT_INIT_DIR2,
  INPUT_NUMBER
 };
 enum {OUTPUT_NUMBER};

 class factory_type: public factory_abstract_type
 {
  int m_hsize, m_vsize;
 public:
  factory_type(int hsize, int vsize):
   m_hsize(hsize), m_vsize(vsize)
  {}
  int number_fsm(void) const { return FSM_NUMBER; }
  int number_input(void) const { return INPUT_NUMBER; }
  int number_output(void) const { return OUTPUT_NUMBER; }
  linkstore_type get_links(void) const
  {
   linkstore_type store;
   // вход - количество циклов
   store.add(
    PSEUDOFSM_NETINPUT, INPUT_CYCLES,
    FSM_TABLE, table_type::INPUT_CYCLES);
   // входные координаты и направление первого шара
   for (int i = 0; i <= INPUT_INIT_DIR1 - INPUT_INIT_X1; i++)
   {
```


выполнением работы.

Поскольку в таких условиях к моменту завершения всех работ, от которых текущая работа зависит, поток должен уже существовать и находится в ожидании, представляется удобным создание сразу всех потоков для всех работ, с тем, чтобы они находились в состоянии ожидания до тех пор, пока не станет допустимо выполнение работы. Однако следует учитывать, что одновременное создание большого количества потоков может чрезмерно нагрузить операционную систему. При этом такая нагрузка может оказаться бесполезной, если, к примеру, задачи окажутся по длительности близкими друг к другу, и такое количество одновременно существующих потоков не будет необходимым.

В связи с этим примем во внимание тот факт, что ни одна работа следующего яруса не может быть начата прежде, чем завершится хотя бы одна работа текущего. В соответствии с таким положением будем создавать потоки поэтапно по одному ярусу, при этом переход к созданию потоков следующего яруса осуществляется только в момент завершения какой-либо работы текущего.

Как и ранее, будем использовать за основу код классов, приведенных в приложении 2. Внешний интерфейс классов снова не изменяется, следовательно, код использующей их программы может оставаться прежним. Более того, как и в предыдущем случае, практически не меняется большинство реализующего классы кода. Изменилась, прежде всего, снова функция jobcomplex_type::run():

```
// выполнение всего комплекса работ
void run(void)
{
 // получим список подлежащих выполнению работ
 joblist_type joblist = get_joblist();

 // получим список зависимостей
 deplist_type deplist = get_deplist();

 // построим ярусно-параллельную структуру номеров работ
 multilevel_type multilevel = build(deplist);
```

```
 // список хэндлов всех потоков
 std::vector<HANDLE> hdl(joblist.size());

 // список всех структур параметров
 std::vector<threadparam_type> thrparam(joblist.size());

 // запустим по очереди каждый ярус работ
 for (int l = 0; l < multilevel.size(); l++)
 {
  std::vector<HANDLE> lasthdl;

  // создадим все потоки текущего яруса
  for (int i = 0; i < multilevel[l].size(); i++)
  {
   // заполняем структуру параметров потока
   int idx = multilevel[l][i];
   thrparam[idx].phdl = &hdl[0];
   thrparam[idx].pdeps = &deplist[idx];
   thrparam[idx].pjob = joblist[idx];

   // создание потока
   DWORD dwId;
   hdl[idx] = ::CreateThread(
    NULL, 0,
    thr_proc, &thrparam[idx],
    0, &dwId);

   // добавляем в список хэндлов текущего яруса
   lasthdl.push_back(hdl[idx]);
  };

  // прежде чем переходить к следующему ярусу,
  // дождемся момента, пока завершится
  // хотя бы один поток текущего яруса
  wait_any(lasthdl);
 };

 // ждем завершения всех оставшихся потоков
 wait_all(hdl);

 // и закрываем хэндлы
 for (int i = 0; i < hdl.size(); i++)
  ::CloseHandle(hdl[i]);
}
```

После построения необходимых структур, создаются список так называемых хэндлов (дескрипторов) потоков и список структур параметров, передаваемых каждому потоку. Список хэндлов потоков необходим для ожидания их завершения. Он заполняется по мере создания потоков, при этом в силу по-


```
 };
 enum {
  OUTPUT_BOUND_LEFT,
  OUTPUT_BOUND_RIGHT,
  OUTPUT_BOUND_TOP,
  OUTPUT_BOUND_BOTTOM,
  OUTPUT_NUMBER
 };
 enum {STATE_ON};

private:
 // размеры стола
 int m_hsize, m_vsize;
 // количество циклов движения
 data_type m_cycles;

 state_type do_out(state_type state)
 {
  // если состояние - включено, отобразим шары
  if (state == STATE_ON)
  {
   // какое-либо отображение стола и шаров
   // ...
   // количество выполненных циклов отображения
   m_cycles++;
  };
  // вычисление следующего состояния
  state = (m_cycles < m_input[INPUT_CYCLES]) ? STATE_ON : STATE_OFF;
  if (state != STATE_OFF)
  {
   // выводим данные о границах
   m_output[OUTPUT_BOUND_LEFT] = 0;
   m_output[OUTPUT_BOUND_RIGHT] = m_hsize + 1;
   m_output[OUTPUT_BOUND_TOP] = 0;
   m_output[OUTPUT_BOUND_BOTTOM] = m_vsize + 1;
  }
  else
   // обнуление выходов влечет остановку шаров
   fill(m_output.begin(), m_output.end(), 0);
  return state;
 }

public:
 table_type(int hsize, int vsize):
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER),
  m_hsize(hsize), m_vsize(vsize), m_cycles(0)
 {
  add_handler(STATE_OFF, handler_type(&table_type::do_out));
  add_handler(STATE_ON, handler_type(&table_type::do_out));
 }
};
```

Приведенный автомат может пребывать в двух состояниях –

в соответствии с описанным выше, после чего происходит проверка наличия препятствия в новой планируемой позиции. Если возможное направление не найдено (к примеру, стол сузился в обоих направлениях или шар прижат в угол другим шаром), шар останавливается и переходит в выключенное состояние. В противном случае происходит изменение текущих координат и направления, а также запись выходных параметров.

Как уже не раз говорилось, приведенная схема для наглядности чрезвычайно упрощена и имеет вследствие этого немало недостатков. В частности, когда между двумя шарами на одной диагонали, двигающихся навстречу друг другу, находится одна пустая клетка, в соответствии с приведенным механизмом они «не заметят» друг друга, и оба встанут на эту клетку, после чего продолжат движение в исходных направлениях.

Безусловно, такое поведение является некорректным и требует внесения дополнительных проверок в код реальной программы. Разумеется, при этом потребуется передача от каждого шара помимо его координат еще и направления движения, что в данном случае было опущено для простоты. Однако здесь для начала потребуется ответить на вопрос, какое поведение обоих шаров являлось бы в такой ситуации корректным. К примеру, оба шара могут встать на пустую клетку, после чего поменять направление, либо же сразу поменять направление, не попадая в клетку и, тем самым, не осуществив столкновения. Оба варианта не самым лучшим образом отражают естественный ход событий, что является следствием упрощения модели и дискретности движения.

Автомат, отвечающий за поведение стола, вследствие своих более простых задач реализуется гораздо проще:

```
class table_type: public fsm_type
{
public:
 enum {
  INPUT_CYCLES,
  INPUT_X1,
  INPUT_Y1,
  INPUT_X2,
  INPUT_Y2,
  INPUT_NUMBER
```


этапности создания нет риска, что какой-либо поток попытается ожидать завершения других потоков по незаполненным хэндлам. Список структур параметров мы храним для того, чтобы быть уверенными, что каждый поток успеет прочитать из него свои параметры прежде, чем структура будет уничтожена.

В теле цикла каждого яруса заполняется соответствующая структура параметров, после чего создается поток. После создания всех потоков текущего яруса, перед переходом к следующему ярусу выполняется ожидание завершения любого из них.

В конце функции выполняется ожидание завершения всех созданных потоков.

Помимо приведенных изменений функции run, в код класса jobcomplex_type добавилось также объявление структуры параметров потока, функции потока и двух функций ожидания:

```
class jobcomplex_type: public job_abstract_type
{
 // ...

private:
 // структура параметров для каждого потока
 struct threadparam_type
 {
  // указатель на массив всех хэндлов
  HANDLE *phdl;

  // указатель на зависимости конкретного потока
  std::vector<int> *pdeps;

  // указатель на работу для выполнения
  job_abstract_type *pjob;
 };

 // ожидание завершения любого потока из переданного списка
 static int wait_any(const std::vector<HANDLE> &hdl)
 {
  assert(hdl.size() > 0);
  DWORD dw = ::WaitForMultipleObjects(
   hdl.size(), &hdl[0], FALSE, INFINITE);
  assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdl.size());
  return dw - WAIT_OBJECT_0;
 }
```

```
 // ожидание завершения всех потоков из переданного списка
 static void wait_all(const std::vector<HANDLE> &hdl)
 {
  if (hdl.size())
  {
   DWORD dw = ::WaitForMultipleObjects(
    hdl.size(), &hdl[0], TRUE, INFINITE);
   assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdl.size());
  };
 }

 // функция, выполняемая потоками
 static DWORD WINAPI thr_proc(LPVOID param)
 {
  threadparam_type &thrp = *static_cast<threadparam_type *>(param);

  // формирование списка хэндлов для ожидания
  // на основе зависимостей текущей работы
  std::vector<HANDLE> hdl;
  for (int i = 0; i < thrp.pdeps->size(); i++)
   hdl.push_back(thrp.phdl[thrp.pdeps->at(i)]);

  // ожидание завершения работ
  wait_all(hdl);

  // выполнение текущей работы
  thrp.pjob->run();

  return 0;
 }

 // ...
};
```

Функция потока формирует на основе переданных полного массива хэндлов и списка зависимостей список хэндлов потоков, завершения которых текущий поток должен дождаться. После выполнения ожидания вызывается функция выполнения работы, и поток завершается.

Внутри функций ожидания в качестве передаваемого адреса массива хэндлов используется адрес первого элемента контейнера vector. Такой подход для чтения массива допустим по той причине, что, в соответствии со спецификацией STL, во-первых, все элементы контейнера vector хранятся в виде массива, во-вторых, операция индексирования возвращает ссылку на соответствующий элемент этого массива.


```
    else
     state = (dx > 0) ? STATE_UP_RIGHT : STATE_DOWN_LEFT;
   };

   // если не выключаемся
   if (state != STATE_OFF)
   {
    // меняем текущие координаты
    m_x += dx;
    m_y += dy;
    // выводим их
    m_output[OUTPUT_X] = m_x;
    m_output[OUTPUT_Y] = m_y;
   }
   else
    fill(m_output.begin(), m_output.end(), 0);
   return state;
  }

public:
 ball_type():
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&ball_type::do_start));
  add_handler(STATE_UP_LEFT, handler_type(&ball_type::do_move));
  add_handler(STATE_UP_RIGHT, handler_type(&ball_type::do_move));
  add_handler(STATE_DOWN_LEFT, handler_type(&ball_type::do_move));
  add_handler(STATE_DOWN_RIGHT, handler_type(&ball_type::do_move));
 }
};
```

Автомат может находиться в пяти состояниях – в четырех состояниях соответственно каждому возможному направлению движения и в выключенном состоянии. В конструкторе класса ball_type каждому из этих состояний задается обработчик. В качестве обработчика выключенного состояния назначается функция do_start, которая выполняет инициализацию шара. Здесь мы опускаем множество проверок входных данных, которые, безусловно, должны быть в реальной программе.

Состояниям движения назначается обработчик do_move. В его задачи входит вычисление следующих координат шара на основе текущих координат, направления движения и наличия препятствий на пути следования. В начале функции do_move вычисляются планируемые изменения текущих координат на основе текущего направления. Далее в цикле по количеству возможных направлений выполняется проверка наличия препятствия в новой позиции. Если препятствие есть, направление движения меняется

```
// текущие координаты
int m_x, m_y;

state_type do_start(state_type state)
{
 // инициализация координат и направления
 m_x = m_input[INPUT_INIT_X];
 m_y = m_input[INPUT_INIT_Y];
 state = m_input[INPUT_INIT_DIR];
 // выводим координаты
 m_output[OUTPUT_X] = m_x;
 m_output[OUTPUT_Y] = m_y;
 return state;
}
state_type do_move(state_type state)
{
 // вычисляем изменения координат
 int dx, dy;
 dx = (state == STATE_UP_LEFT || state == STATE_DOWN_LEFT)? -1 : 1;
 dy = (state == STATE_UP_LEFT || state == STATE_UP_RIGHT) ? -1 : 1;

 // ищем возможное направление
 bool found = false;
 for (int i = 0; !found && i < DIR_NUMBER; i++)
 {
  // проверим возможность столкновения с другим шаром
  if (m_x + dx == m_input[INPUT_OTHER_X] &&
   m_y + dy == m_input[INPUT_OTHER_Y])
  {
   // меняем направление на противоположное
   dx *= -1;
   dy *= -1;
  }
  // проверим возможность столкновения с границами
  else if (m_x + dx <= m_input[INPUT_BOUND_LEFT])
   dx *= -1;
  else if (m_x + dx >= m_input[INPUT_BOUND_RIGHT])
   dx *= -1;
  else if (m_y + dy <= m_input[INPUT_BOUND_TOP])
   dy *= -1;
  else if (m_y + dy >= m_input[INPUT_BOUND_BOTTOM])
   dy *= -1;
  else
   found = true;
 };

 // останавливаемся, если двигаться некуда
 if (!found)
  state = STATE_OFF;
 else
 {
  if (dx * dy > 0)
   state = (dx > 0) ? STATE_DOWN_RIGHT : STATE_UP_LEFT;
```


Глава 3. Сети конечных автоматов

В текущей главе мы рассмотрим схемы создания параллельных программ на основе сетей конечных автоматов. Основа параллелизма такого подхода заключается в том, что каждый автомат в сети является автономным элементом и не связан с другими автоматами, кроме как входными и выходными значениями на каждом такте. Таким образом, на протяжении каждого такта все автоматы могут работать параллельно.

Разумеется, для реализации такого подхода придется прибегнуть к методам автоматного программирования (Automata-Based Programming, State-Based Programming) для преобразования алгоритма программы к автоматному виду. Существует немало работ, посвященных этой теме, в частности, рассмотрению SWITCH-технологии [24, 25, 29], КА-технологии [19] и преобразованию алгоритмов в автоматные [28, 30], поэтому мы не будем останавливаться на ней. Скажем лишь, что есть немало областей программирования, в которых автоматный подход может существенно облегчить проектирование и реализацию.

3.1 Программирование конечных автоматов

Изложим вкратце основы автоматного программирования. Описания конечного автомата (Finite-State Machine, Finite-State Automaton) разнятся в деталях от издания к изданию в зависимости от области применения, в частности, от ориентации на программную или аппаратную реализацию. Более того, описания автомата различаются даже при ориентации на программную реализацию, особенно среди сравнительно недавних публикаций [9, 13].

Мы здесь приведем далеко не новое очень обобщенное описание конечного автомата [22], достаточное для понимания принципов создания сетей конечных автоматов для применения в параллельном программировании. Отметим также, что автомат Мура считается более удобным для аппаратной реализации [41], для программной же нет явных предпочтений, однако чаще используется автомат Мили или смешанный авто-

мат Мура-Мили [27].

Конечный автомат характеризуется следующими параметрами [22]:

- множество состояний $\{S_0, S_1, \ldots, S_N\}$;
- множество совокупностей входных значений (входной алфавит);
- множество совокупностей выходных значений (выходной алфавит);
- начальное состояние S_0 (в котором автомат находится перед началом работы);
- конечное состояние (или множество таковых), т.е. такое состояние, по достижении которого работа автомата считается завершенной (зачастую удобно использовать начальное состояние S_0);
- таблица переходов (зависимость следующего состояния от текущего состояния и текущей совокупности входных значений);
- таблица выходов (зависимость совокупности текущих выходных значений от текущего состояния и текущей совокупности входных значений).

Удобство совпадения конечного и начального состояний заключается в возможности повторного использования автомата.

Таблицы, указанные в последних двух пунктах, при некоторых допущениях полностью описывают все остальные перечисленные характеристики автомата. К примеру, на рис. 15 мы видим таблицы переходов и выходов некоторого автомата. Из таблиц видно множество состояний $\{A, B, C\}$, множество входов $\{00, 01, 10, 11\}$ и множество выходов $\{'a', 'b', 'c', 'd', 'e'\}$. Если оговориться, что в первом столбце таблицы всегда описывается начальное/конечное состояние, известны и они – $S_0 = A$. Входом такого автомата может быть, к примеру, двухбитное целое число или два логических значения, выход – символьный.

и направлении движения, данные о границах стола и текущие координаты другого шара. Передача информации о границах стола в явном виде на каждом такте позволяет использовать стол с двигающимися границами, хотя мы эту возможность использовать не будем.

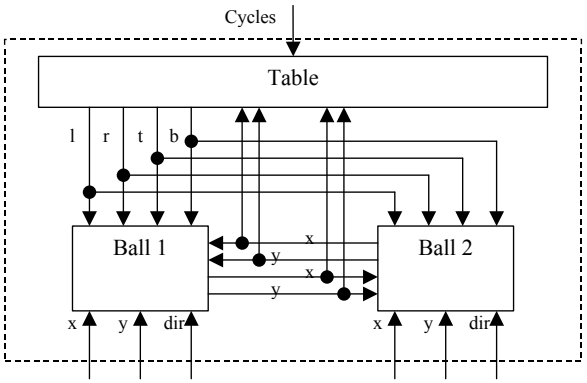

Рис. 21

Код класса автомата, реализующего поведение шара, может быть следующим:

```
class ball_type: public fsm_type
{
public:
 enum {
  INPUT_INIT_X,
  INPUT_INIT_Y,
  INPUT_INIT_DIR,
  INPUT_BOUND_LEFT,
  INPUT_BOUND_RIGHT,
  INPUT_BOUND_TOP,
  INPUT_BOUND_BOTTOM,
  INPUT_OTHER_X,
  INPUT_OTHER_Y,
  INPUT_NUMBER
 };
 enum {OUTPUT_X, OUTPUT_Y, OUTPUT_NUMBER};
 enum {
  STATE_UP_LEFT,
  STATE_UP_RIGHT,
  STATE_DOWN_LEFT,
  STATE_DOWN_RIGHT,
  DIR_NUMBER
 };

private:
```

ры могут отталкиваться от границ стола и сталкиваться друг с другом. При этом, разумеется, направление движения шаров меняется. Удобство реализации такой задачи в виде автоматной сети основывается на том, что в каждый момент времени шару для принятия решения о направлении дальнейшего движения необходима информация о своем текущем направлении, а также о направлении движения и координатах других шаров.

Поскольку в наши цели входит лишь иллюстрация вариантов использования автоматов и сетей, мы снимем множество условий, приближающих задачу к реальности, чем сильно упростим задачу и сделаем ее реализацию более наглядной. Реализуем задачу для случая стола с двумя шарами. Будем считать, что шары двигаются по клеткам под углом 45 градусов к границам стола в одном из четырех возможных в этом случае направлений. При столкновении с границей стола каждый шар меняет направление естественным образом. При столкновении шаров между собой оба продолжают двигаться дальше в противоположном направлении (рис. 20).

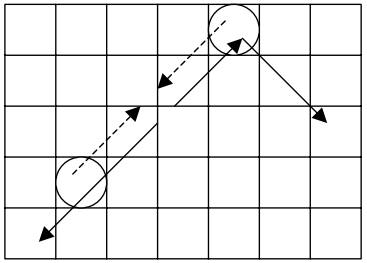

Рис. 20

Будем реализовывать автоматную сеть в виде трех автоматов (рис. 21). Один автомат играет роль стола. В его задачи входит получение данных о текущих координатах шаров и соответствующее отображение какими-либо средствами. Также на входе стол принимает количество циклов выполнения. На выходе стол выдает информацию о своих границах – левой, правой, верхней и нижней. Еще два автомата играют роль шаров, каждый из которых принимает инициализационные данные о своих координатах

State	A	B	C
00	B	C	C
01	B	A	B
10	C	C	A
11	B	B	B

Out	A	B	C
00	a	a	d
01	b	c	e
10	d	d	a
11	c	b	c

Рис. 15

Помимо таблиц, описание автомата также может быть представлено диаграммой состояний (графом переходов). Вершины такого графа представляют весь набор возможных состояний автомата; направленные дуги, характеризующие переходы, помечаются входными данными, соответствующими переходу, и выходными данными автомата при такой ситуации (условием перехода и действием на переходе). К примеру, на рис. 16 представлен граф переходов конечного автомата, описанного таблицами на рис. 15.

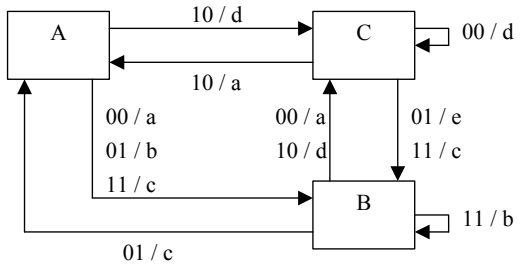

Рис. 16

Сказанное выше касалось скорее математического описания конечных автоматов. Применительно же к программной реализации следует добавить следующее. Жизненный цикл автомата представляется программным циклом со сменой состояний от начального до конечного. В теле цикла на каждом такте происходит проверка текущего состояния и входных данных, после чего осуществляется возможное изменение состояния и генерация выходных данных. В программе, разумеется, могут быть дополнительные действия, характерные для предметной области. К примеру, в управляющей программе могут быть выполнены вызовы

соответствующих функций управления оборудованием.

Также в течение такта работы могут быть выполнены модификации каких-либо внутренних переменных автомата, даже в случае сохранения предыдущего состояния. Фактически, в соответствии с приведенным выше описанием автомата, это является сменой состояния. Однако с целью упрощения программирования и повышения наглядности принципов функционирования программы таким путем производится «группировка» большого количества схожих состояний как одного. К примеру, если автомат в цикле последовательно производит обработку элементов массива, производится смена состояний «Обработка элемента 0», «Обработка элемента 1», «Обработка элемента 2» и т.д. Однако в программе мы можем себе позволить хранение текущего индекса обрабатываемого элемента отдельно от состояния, объединив все перечисленные состояния в единое состояние «Обработка массива», тем самым существенно упростив и программирование, и читабельность программы.

Буквы входного алфавита программного автомата могут быть не только заданы извне, но также могут быть вычислены самим автоматом на основе результатов каких-либо инкапсулированных автоматом действий. К примеру, могут быть считаны из конфигурационных файлов или базы данных.

В качестве иллюстрации приведем простой вариант реализации программного автомата, диаграмма состояний которого изображена на рис. 16.

```
char state = 'A';
do
{
 bool in1, in2;
 char out;
 get_input(in1, in2);
 switch (state)
 {
 case 'A':
  out = in1 ? (in2 ? 'c' : 'd') : (in2 ? 'b' : 'a');
  state = (in1 && !in2) ? 'C' : 'B';
  break;
 case 'B':
  out = in1 ? (in2 ? 'b' : 'd') : (in2 ? 'c' : 'a');
  state = (!in2) ? 'C' : (in1 ? 'B' : 'A');
  break;
```



```
    int rank;
    MPI_Comm_rank(m_fsmnet.m_commlocal, &rank);
    if (rank == 0)
     copy(input.begin(),input.end(),m_data.begin()+m_outpos.back());
    // и рассылается остальным
    MPI_Bcast(
     &m_data[m_outpos.back()], m_outsize.back(), MPI_INT,
     0, m_fsmnet.m_commlocal);
   };
  }
  // ...
};
```

Обе функции выполняют полезную работу лишь в случае, когда текущий процесс является локальным участником в текущей сети, т.е. мастером нижележащего автомата. В остальном функция put_output отличается от приведенной выше для простой реализации одноуровневой сети лишь коммуникатором. Функция же put_input сохраняет переданные входные данные сети лишь в случае, когда текущий процесс является мастером текущей сети, т.е. нулевым среди всех процессов переданного извне коммуникатора m_commfull. Этот же процесс по построению является нулевым для коммуникатора m_commlocal. После сохранения входных данных сети они должны быть разосланы всем процессами локальных участников.

Приведенное описание модифицированных классов позволяет строить распределенные автоматные сети с многоуровневым вложением и внутренним распараллеливанием элементарных автоматов.

3.4 Бильярдные шары

Рассмотрим теперь варианты реализации конкретных задач с использованием приведенных классов. Задача, выбранная в качестве первого примера, названа бильярдом, хотя на самом деле упрощена настолько, что называться так не должна. Тем не менее, эта задача даже в таком упрощении весьма подходит для автоматной реализации. Более того, при попытке реализации ее классическими методами, так или иначе, в большинстве случаев будет получен алгоритм, близкий к автоматному.

Реализуемая задача заключается в отображении некоего стола, по которому без потери скорости катаются шары. Эти ша-

Поскольку проверка выключения производится во всех процессах сети, результат рассылается мастером в контексте коммуникатора m_commfull.

Наконец, изменения должны коснуться функций помещения выходных данных автомата и входных данных сети в область общих данных сети. В обеих функциях используется коммуникатор локальных участников сети, который с этой целью должен быть каким-либо образом передан конструктору класса shared_area_type:

```
class shared_area_type
{
 // ...
private:
 // ссылка на сеть-владельца
 fsmnet_type &m_fsmnet;

public:
 // ...
 shared_area_type(
  const std::vector<link_type> &links, fsmnet_type &fsmnet):
  m_fsmnet(fsmnet)
 {
  // ...
 }
 // сохранение выходных данных конкретного автомата
 void put_output(int i, const std::vector<data_type> &output)
 {
  // рассмотрению подлежат только локальные участники
  if (m_fsmnet.m_commlocal != MPI_COMM_NULL)
  {
   // синхронизация всех автоматов - для отделения от чтения входов
   MPI_Barrier(m_fsmnet.m_commlocal);
   copy(output.begin(), output.end(), m_data.begin() + m_outpos[i]);
   // выполним полный обмен - раскидаем все выходы всем процессам
   // в конце массивов размеров и смещений есть по одному
   // лишнему элементу, но это не мешает - они не учитываются
   MPI_Allgatherv(
    &m_data[m_outpos[i]], m_outsize[i], MPI_INT,
    &m_data[0], &m_outsize.front(), &m_outpos.front(), MPI_INT,
    m_fsmnet.m_commlocal);
  };
 }
 // сохранение входных данных сети
 void put_input(const std::vector<data_type> &input)
 {
  // рассмотрению подлежат только локальные участники
  if (m_fsmnet.m_commlocal != MPI_COMM_NULL)
  {
   // вход сети принимается только нулевым процессом в сети
```



```
 case 'C':
  out = in1 ? (in2 ? 'c' : 'a') : (in2 ? 'e' : 'd');
  state = (in2) ? 'B' : (in1 ? 'A' : 'C');
  break;
 };
 put_output(out);
} while (state != 'A');
```

Работа автомата представляется циклом, условие завершения которого – достижение начального состояния. Текущее состояние автомата хранится в его внутренней переменной. В теле цикла на каждой итерации (каждом такте) производится чтение входных значений, после чего на основе текущего состояния и считанных значений формируются и выводятся выходные значения автомата.

Автоматное программирование оказывается очень удобным при реализации широкого класса задач, включающего программы логического управления, компиляторы, игры. Однако существуют задачи, для которых автоматное программирование может лишь утяжелить процесс реализации. Такой подход оказывается сродни попытке воплотить какой-нибудь сложный вычислительный алгоритм на машине Тьюринга. Написание вычислительной процедуры классическим путем, включая отладку, может оказаться на порядок менее трудоемким, нежели преобразование алгоритма в автоматный, при этом сохранится большее соответствие между исходным описанием алгоритма математическими формулами и программным представлением. Хорошим примером тому является представленная в этом разделе автоматная реализация алгоритма суммирования сдваиванием.

Любая среда/средство/язык программирования, предназначенная для реализации каких-либо задач, должна, прежде всего, предоставлять простоту и наглядность описания стоящей задачи. Далеко не все алгоритмы описываются автоматами наглядно, несмотря на то, что, безусловно, в принципе, могут быть ими описаны.

Одной из важных особенностей автоматного программирования является возможность визуализации алгоритма с последующей генерацией программного кода без участия программиста. К примеру, если характеристики автомата уже описаны таб-

лицами переходов и выходов или графически диаграммой состояний, программный код такого автомата может быть сгенерирован автоматически соответствующими программными средствами. На сегодняшний день существует немало подобных средств генерации (к примеру, FSMDesigner, FSMGenerator и т.п.), однако мы не будем на них останавливаться.

Считается, что автоматные программы практически не требуют отладки, либо требуют ее в гораздо меньшей степени [4, 13, 19]. Отчасти это так. При классическом подходе алгоритм решения поставленной задачи, порой, пишется «на ходу», т.е. в процессе реализации много раз меняется. Тот исходный факт, что алгоритм решения задачи должен существовать в виде блок-схемы еще до начала процесса программирования, в реальности практически всегда игнорируется. В результате такого создания алгоритмов последние зачастую не имеют доказательной основы и оказываются нерабочими при некоторых исходных данных. В случае же автоматного программирования создание алгоритма в принудительном порядке отодвигается на ранние этапы разработки. Для преобразования некоторого алгоритма в автоматный исходный алгоритм должен уже быть создан и отлажен. Разумеется, после трансляции полученного таким образом автоматного алгоритма на формальный язык даже силами программиста программа будет содержать гораздо меньше ошибок, нежели при формировании алгоритма в процессе реализации. При программной же трансляции ошибок удается избежать практически полностью.

3.2 Параллелизм сетей конечных автоматов

Сами по себе реализованные программно конечные автоматы являются последовательными и, вопреки мнению некоторых авторов, в общем случае потенциального параллелизма в себе не несут. Параллелизм может содержаться при вычислении условий перехода, однако для произвольного автоматного алгоритма далеко не всегда это позволит повысить эффективность вычислений. Зачастую такое распараллеливание может оказаться просто дублированием одних и тех же вычислений во всех параллельных

текущего автомата. Всего таких коммуникаторов три – также по количеству автоматов.

Наконец, в конце работы конструктора fsmnet_type вызывается создание нижележащего автомата с только что созданным коммуникатором m_commlower в качестве параметра, который должен быть передан создаваемому объекту класса fsmnet_type с целью последующего разбиения на коммуникаторы или объекту класса fsm_type с целью использования для внутреннего распараллеливания.

Деструктор fsmnet_type выполняет уничтожение нижележащего автомата, после чего производит освобождение созданных в конструкторе коммуникаторов. Коммуникатор m_commlocal освобождают только те процессы, для которых он действителен, т.е. только те, которые являются мастерами нижележащих автоматов (имеют нулевой ранг в группе коммуникатора m_commlower).

Функция do_work класса fsmnet_type полностью аналогична той, что была приведена выше для случая простого распараллеливания с использованием MPI одноуровневой автоматной сети. Функция же проверки выключения сети, в отличие от предыдущего случая, опрашивает на предмет выключения лишь локальных участников, причем результат получает лишь мастер-процесс текущей сети:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 bool is_off(void) const
 {
  // сеть в начальном состоянии, когда все автоматы в начальном
  int isoff = m_pfsm.back()->is_off();
  // сольем статусы от локальных участников в нулевой процесс
  int allisoff;
  if (m_commlocal != MPI_COMM_NULL)
   MPI_Reduce(
    &isoff, &allisoff, 1, MPI_INT, MPI_LAND, 0, m_commlocal);
  // и распределим результат между всеми
  // по построению нулевой в m_commfull тот же, что и в m_commlocal
  MPI_Bcast(&allisoff, 1, MPI_INT, 0, m_commfull);
  return allisoff;
 }
 // ...
};
```

ответственный за «представительство» нижележащего автомата в текущей сети. Этот же процесс (мастер) ответственен за передачу входных данных нижележащего абстрактного автомата всем его процессам и изъятие из него выходных. Все такие процессы текущей сети объединяются в группу так называемых локальных участников. Список номеров процессов локальных участников формируется в процессе опроса размеров автоматов. На основе созданной группы локальных участников создается коммуникатор m_commlocal, в контексте которого будет производиться взаимодействие в рамках обмена выходными данными между абстрактными автоматами текущей сети.

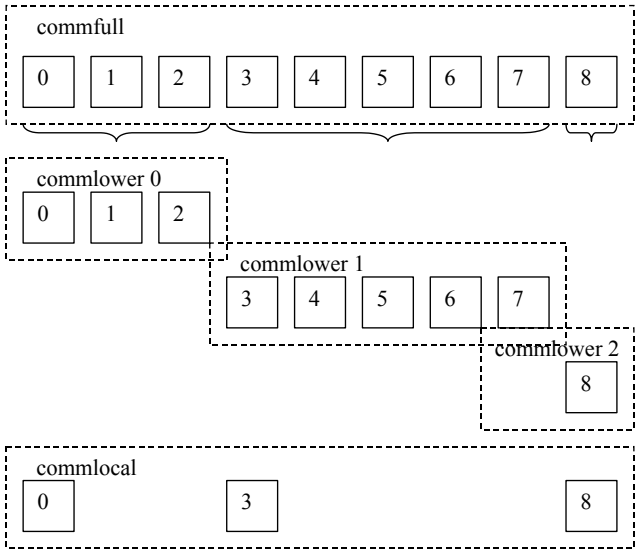

Рис. 19

Пример распределения процессов по коммуникаторам можно увидеть на рис. 19. Здесь показано, как девять процессов распределяются на три автомата, размеры которых равны соответственно трем, пяти и одному. Коммуникатор m_commfull содержит все девять процессов, коммуникатор m_commlocal содержит три процесса – по количеству автоматов. Наконец, коммуникатор m_commlower содержит количество процессов, равное размеру

ресурсах, будь то процессы или потоки. Однако ситуация совершенно иная в отношении сетей конечных автоматов.

Сетью конечных автоматов называют некоторое количество связанных между собой входными и выходными данными (сигналами) конечных автоматов. Вход каждого автомата сети может быть подключен к входу сети или выходу другого автомата. Выход каждого автомата может быть подключен, соответственно, к входу другого автомата или выходу сети. К одному источнику сигнала (выходу автомата или входу сети) может быть подключено более одного приемника сигнала (входа автомата или выхода сети). К одному же приемнику сигнала может быть подключен только один источник. Пример связи автоматов в некоторой автоматной сети можно увидеть на рис. 17. Здесь изображена сеть из трех автоматов с двумя входам и четырьмя выходами. Каждый из автоматов имеет по четыре входа, при этом два из них имеют три выхода, еще один – два выхода.

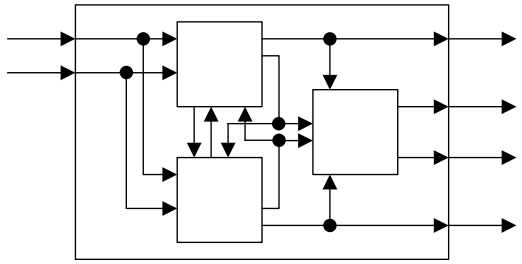

Рис. 17

С точки зрения функционирования сеть автоматов также представляет собой некий сложный автомат, состояние которого в каждый момент времени определяется совокупностью состояний внутренних автоматов сети. Таким образом, возможно построение автоматных сетей с многоуровневым вложением. При программной реализации такое вложение может иметь смысл по различным причинам. К примеру, такой подход может позволить создание сложных распараллеленных базовых элементов на основе элементарных. Еще одна причина целесообразности такого

вложения будет описана ниже при описании примера программной реализации.

Все автоматы сети работают синхронно, т.е. начало работы следующего такта в каждом автомате не происходит до тех пор, пока не завершится работа предыдущего такта во всех автоматах. На протяжении каждого такта, т.е. между моментами чтения входных сигналов и записи выходных, все автоматы работают независимо друг от друга, поэтому могут выполнять работу параллельно.

Программно реализованная автоматная сеть, в отличие от аппаратной, не нуждается во внешней генерации синхроимпульсов, поэтому длительность такта в большинстве случаев не является постоянной величиной. Естественным способом синхронизации параллельно работающих автоматов является установка барьера. В такой ситуации длительность такта работы сети определяется максимальной длительностью текущего такта среди всех автоматов сети. Иными словами, такт работы сети завершается в тот момент, когда закончит обработку такта последний автомат сети.

Реализация параллельной программы сетью конечных автоматов имеет смысл лишь в том случае, когда такт занимает достаточно много времени по сравнению с накладными расходами на передачу данных между автоматами и обеспечение синхронизации тактов.

В частности, представленный ниже алгоритм суммирования сдваиванием носит лишь иллюстративный характер, поскольку накладные расходы на действия, осуществляемые для обеспечения работы всех вычисляющих автоматов на одном такте, существенно выше длительности полезного действия, выполняемого каждым таким автоматом на одном такте (суммирование двух чисел).

Для более подробного ознакомления с теорией конечных автоматов рекомендуется обратиться к специально посвященной этой теме литературе [8, 22]. Помимо учебников, также доступно много достаточно современных статей [4, 13, 17, 18, 25, 27, 29]. В частности, немало публикаций можно найти в Internet на страни-


```
  // зачистка созданных коммуникаторов
  MPI_Comm_free(&m_commlower);
  if (m_commlocal != MPI_COMM_NULL)
   MPI_Comm_free(&m_commlocal);
 }
 // ...
};
```

К конструктору fsmnet_type, также как и к конструктору fsm_type, добавляется дополнительный параметр, характеризующий коммуникатор, в рамках которого будут работать все нижележащие автоматы текущей сети. Объектом автомата, унаследованным от fsm_type, переданный коммуникатор может быть использован по своему усмотрению. Конструктор сети самого верхнего уровня вызывается с каким-либо определенным вне сети коммуникатором с нужным количеством процессов, т.е. равным полному размеру всей сети. К примеру, это может быть коммуникатор MPI_COMM_WORLD.

Чтобы понять, какую работу в приведенном коде выполняет конструктор, следует оговорить, как происходит распределение автоматов по процессам. Тот коммуникатор текущей сети, в контексте которого будут обмениваться выходными данными ее абстрактные автоматы (элементарные автоматы или вложенные сети), должен включать ровно столько процессов, сколько абстрактных автоматов она непосредственно содержит, т.е. сколько возвращается функцией number_fsm фабрики сети. В то же время, каждому абстрактному автомату должен быть передан свой коммуникатор, включающий количество процессов, равное размеру этого автомата. С этой целью все процессы переданного через конструктор коммуникатора m_commfull разделяются на группы, каждая из которых соответствует одному из абстрактных автоматов сети. Для выполнения этой операции путем опроса фабрики сети на предмет размеров ее автоматов предварительно выясняется номер m_fsmnum нижележащего абстрактного автомата текущей сети, который будет реализовываться данным процессом. После этого коммуникатор m_commfull разделяется на коммуникаторы m_commlower по значению номера текущего нижележащего автомата m_fsmnum.

Из каждой созданной группы выделяется нулевой процесс,

```
{
 // ...
private:
 // номер автомата текущей сети, реализуемого нашим процессом
 int m_fsmnum;
 // полный коммуникатор по всем процессам сети
 MPI_Comm m_commfull;
 // коммуникатор локальной связи по сети автоматов
 MPI_Comm m_commlocal;
 // коммуникатор, переданный нижней сети/автомату
 MPI_Comm m_commlower;

public:
 fsmnet_type(factory_abstract_type &factory, MPI_Comm commfull):
  m_factory(factory), m_shared(m_factory.get_links().get(), *this),
  m_commfull(commfull)
 {
  // заполним массив локальных участников (нижних мастеров)
  std::vector<int> loc;
  // и локализуем свою принадлежность конкретному нижнему автомату
  int rank;
  MPI_Comm_rank(m_commfull, &rank);
  m_fsmnum = - 1;
  for (int i = 0, cur = 0; i < m_factory.number_fsm(); i++)
  {
   // положим индекс первого процесса очередного автомата
   loc.push_back(cur);
   // перейдем к следующему автомату
   cur += m_factory.fsm_size(i);
   // запомним индекс последнего автомата, если он наш
   if (m_fsmnum < 0 && rank < cur)
    m_fsmnum = i;
  };

  // создадим коммуникатор локальных участников сети
  MPI_Group group, newgroup;
  MPI_Comm_group(m_commfull, &group);
  MPI_Group_incl(group, loc.size(), &loc[0], &newgroup);
  MPI_Comm_create(m_commfull, newgroup, &m_commlocal);
  MPI_Group_free(&newgroup);
  MPI_Group_free(&group);

  // создадим нижний коммуникатор (для подсети или автомата)
  MPI_Comm_split(m_commfull, m_fsmnum, rank, &m_commlower);

  // создадим нижний автомат или подсеть
  m_pfsm.push_back(m_factory.create_fsm(m_fsmnum, m_commlower));
 }

 ~fsmnet_type(void)
 {
  // уничтожение автомата
  m_factory.destroy_fsm(m_fsmnum, m_pfsm.back());
```


це http://www.softcraft.ru/auto.shtml.

3.3 Пример программной реализации

Здесь для иллюстрации мы рассмотрим один из вариантов программной реализации автоматов в некотором обобщенном виде, пригодном для использования при реализации достаточно широкого класса конечно-автоматных сетей. Прежде, чем перейти к обсуждению исходного текста, необходимо оговорить некоторые требующие учета при описании предлагаемой программной реализации моменты.

Прежде всего, в автоматной сети должна быть общая промежуточная область «выходов», т.е. единая общая область памяти, куда попадают все значения выходных сигналов автоматов после каждого такта, и откуда эти значения будут считаны перед следующим тактом. Эта область должна содержать именно «выходы» автоматов, а не «входы», вследствие типа связи «один выход – много входов». В противном случае возникают необходимость дублирования информации и дополнительные сложности реализации.

Во избежание перезаписи непрочитанных данных чтение входных данных всеми автоматами должно быть жестко отделено от записи выходных данных всеми автоматами. В нашем случае действия автомата разделены на три функции – чтение входных данных, как таковые внутренние действия автомата и запись выходных данных. Интерпретация данных, вычисление локальных событий, действие на переходе и переход объединены в одну функцию между чтением и записью данных. На интервале между чтением входов и записью выходов необходим лишь один барьер, поэтому функция действия может быть выполнена без барьера как сразу после чтения, так и непосредственно перед записью. В нашем случае действие автомата выполняется сразу после чтения.

Будем предполагать в нашем примере, что все входные и выходные данные автоматов одного типа – целочисленные. Общая область памяти для значений выходов автоматов строится программно на основе описаний автоматов и связей их между со-

бой. При этом может быть осуществлена проверка корректности связей и соответствия их набору автоматов. Каждая связь в наборе может быть представлена в одной из трех форм:

$$(NET_IN, index) \to (FSM_num, index);$$

$$(FSM_num, index) \to (NET-OUT, index);$$

$$(FSM_num, index) \to (FSM_num, index).$$

Здесь первый элемент каждой пары в скобках характеризует вход сети, выход сети либо конкретный автомат сети. Второй элемент каждой пары характеризует индекс конкретного сигнала (входа или выхода) для объекта, указанного первым элементом пары. При проверке корректности связей должны учитываться следующие факторы:

- одному входу автомата или выходу сети соответствует только один выход автомата или вход сети;
- вход сети может быть только в левой части связи;
- выход сети может быть только в правой части связи.

При проверке соответствия набору автоматов может быть осуществлена проверка границ номеров входов и выходов (они должны соответствовать их количеству для каждого автомата или сети в целом), а также может быть выполнена проверка отсутствия «висящих концов» (отсутствия выходов, не присоединенных ни к одному входу, и наоборот).

Однако в случае реализации распределенной автоматной сети выполнение проверки соответствия набору автоматов оказывается неэффективным вследствие рассредоточенности автоматов между узлами и, как следствие, сравнительно высоких затратах на получение информации о количестве входов и выходов. Поскольку такая проверка требуется, как правило, лишь на этапе отладки автоматного алгоритма, ее можно реализовать в рамках последовательной программы.

При формировании общей области выходных данных для распределенной автоматной сети удобнее всего ориентироваться только на набор связей, поскольку при условии отсутствия «висящих концов» его вполне достаточно для вычленения информации о количестве выходов и входов автоматов и, соответственно,

fsm_abstract_type. Однако такому подходу препятствует тот факт, что для выяснения размера конкретного автомата потребуется его создание, причем далеко не только в тех процессах, в которых он действительно должен быть создан для последующего функционирования.

Информация о требуемом размере автомата должна быть известна фабрике сети. Поскольку ей по определению известно соответствие номеров автоматов конкретным автоматам, это не составляет проблему. При создании внутренней сети ее конструктору также должен быть передан объект фабрики внутренней сети. Он может содержаться внутри объекта фабрики внешней сети, тогда информация о размере внутренней сети также будет доступна фабрике внешней. Таким образом, объект фабрики внешней сети удобно представляется «деревом» фабрик сети, т.е. содержит фабрики всех внутренних сетей (рис. 18). Именно по этой причине, поскольку такой объект дерева фабрик должен содержаться в каждом процессе, все классы фабрик должны быть «легкими» для создания. Затраты вычислительных ресурсов должны требоваться лишь при вызове функций создания/уничтожения конкретного автомата или получения списка связей.

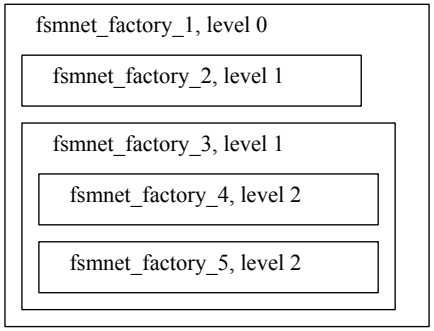

Рис. 18

На основе полученной от фабрики сети информации о размерах внутренних автоматов конструктор fsmnet_type выполняет создание необходимых коммуникаторов:

```
class fsmnet_type: public fsm_abstract_type
```

будем приводить лишь подвергнувшиеся изменению фрагменты кода, приведенного в приложении 3.

Требуемые изменения предоставляемого классами интерфейса диктуются следующими обстоятельствами. Каждой автоматной сети для выполнения обмена данными между ее внутренними автоматами должен быть сопоставлен свой коммуникатор. Поскольку рассматриваемая версия MPI предполагает статическое существование групп процессов (отсутствие возможности изменения группы после создания) перед созданием коммуникатора необходимо знать, сколько процессов он должен содержать. С этой целью к классу фабрики сети добавляется еще одна функция:

```
// абстрактный тип фабрики автоматной сети
class factory_abstract_type
{
public:
 // ...
 // полное количество параллелей в автомате
 virtual int fsm_size(int i) const = 0;
 // ...
 // создание i-го автомата в сети
 virtual fsm_abstract_type *create_fsm(int i, MPI_Comm commfull) =0;
 // ...
};
```

Добавленный параметр в функции создания автомата нужен для передачи коммуникатора конструктору fsmnet_type или fsm_type, о чем подробнее будет сказано позже. Новая функция fsm_size возвращает требуемое количество параллельных ресурсов для каждого автомата сети. Будем в дальнейшем называть это число размером автомата. Поскольку автомат может быть автоматной сетью, количество выполняющих его работу процессов может быть более одного. Более того, любой автомат, даже не являясь сетью, может требовать более одного процесса. К примеру, у какого-либо автомата на одном такте работы может выполняться полноценный цикл работы некоторой другой, не связанной с данной, автоматной сети, или каким-либо другим образом организованная параллельная работа.

Разумеется, дополнительную функцию, возвращающую размер автомата, логичнее было бы внести не в интерфейс фабрики сети factory_abstract_type, а в интерфейс автомата

построения общей области памяти. Это позволит исключить межпроцессные коммуникации для определения количества входов и выходов каждого автомата.

Во избежание ошибок при указании связей удобнее всего их генерировать автоматически путем интерпретации графического представления сети. Для этой цели, как и для начальной генерации программного описания конкретного автомата, могут быть созданы специальные программные средства, подобные описанным в [4].

Вложение сетей, помимо очевидных преимуществ модульного построения, имеет смысл также для локализации связей. При универсальной (не привязанной к структуре конкретной автоматной сети) реализации параллельно работающей сети в системах с распределенной памятью после каждого такта потребуется рассылка выходных данных каждого автомата всем остальным автоматам сети. Зачастую это нецелесообразно, поскольку не всем автоматам нужны результаты работы всех других автоматов. Иначе говоря, выходы конкретного автомата могут требоваться на входе лишь небольшому количеству других автоматов сети, остальным же рассылка его выходных значений не требуется. В таких ситуациях бывает удобно объединить независимые группы автоматов в отдельные сети, не затрагивая при этом универсальность реализации сети. Каждая независимая группа автоматов обменивается данными с другими автоматами своей группы, не производя ненужного обмена с остальными автоматами.

Изначально автомат находится в выключенном состоянии, которое является начальным и конечным. При завершении работы автомат должен перейти в начальное состояние. Будем считать, что сеть завершила свою работу, когда завершили работу, т.е. перешли в начальное состояние, все автоматы сети.

Также будем считать, что если автомат завершил работу, он не выдает на выходе никаких данных. Это дополнение позволит избежать потерь данных при проверке завершения работы сети автоматов. Иначе возможна ситуация, когда один автомат сети, будучи уже выключенным, передает данные другому автомату, еще не включенному, при этом работа сети завершается, хотя ал-

горитм еще не отработал.

В момент старта все автоматы пребывают в выключенном состоянии, поэтому сигналы внутри сети между выходами одних автоматов и входами других отсутствуют.

В некоторых случаях проверять факт завершения работы всеми автоматами сети оказывается чересчур накладно, поэтому может быть удобным при параллельных вычислениях переопределять процедуру проверки завершения работы сети более простыми средствами, нежели коммуникации всех процессов. К примеру, проверка завершения может осуществляться путем анализа одного из выходов сети, предназначенного для указания наличия финального результата работы сети. Здесь, правда, следует оговориться, что, при введенных выше условиях, таким путем может быть выявлен не факт завершения работы сети, а лишь тот факт, что сеть завершит работу на следующем такте, поскольку после завершения работы сеть не выдает никаких выходных значений.

Далее рассмотрим конкретные варианты параллельной реализации с использованием описанного подхода. Позже будут приведены примеры построения конкретных автоматных сетей с использованием описанных общих классов.

3.3.1 Реализация с использованием OpenMP

В приложении 3 приведен исходный текст последовательной реализации базовых классов конечного автомата и автоматной сети. Класс автоматной сети распараллелен с помощью директив OpenMP. Оба этих класса унаследованы от абстрактного класса конечного автомата (fsm_abstract_type), содержащего шесть функций:

```
// абстрактный тип автомата
class fsm_abstract_type
{
public:
 // тип входных и выходных данных
 typedef int data_type;
 // тип состояния автомата
 typedef int state_type;

 // количество входных каналов
 virtual int number_input(void) const = 0;
```


выходных данных разных размеров от всех автоматов в одну область и копирование полученной области в адресное пространство всех процессов коммуникатора MPI_COMM_WORLD. Помимо этого, для предотвращения возможности записи выходов до выполнения чтения входов в начало функции добавлена барьерная синхронизация всех процессов коммуникатора:

```
class shared_area_type
{
 // ...
public:
 // сохранение выходных данных конкретного автомата
 void put_output(int i, const std::vector<data_type> &output)
 {
  // синхронизация всех автоматов - для отделения от чтения входов
  MPI_Barrier(MPI_COMM_WORLD);
  copy(output.begin(), output.end(), m_data.begin() + m_outpos[i]);
  // выполним полный обмен - раскидаем все выходы всем процессам
  // в конце массивов размеров и смещений есть по одному
  // лишнему элементу, но это не мешает - они не учитываются
  MPI_Allgatherv(
   &m_data[m_outpos[i]], m_outsize[i], MPI_INT,
   &m_data[0], &m_outsize.front(), &m_outpos.front(), MPI_INT,
   MPI_COMM_WORLD);
 }
 // ...
};
```

Модифицированный таким образом набор классов потребует минимальных изменений в использующем их коде. Необходимыми из них являются лишь вставки вызовов MPI_Init и MPI_Finalize.

3.3.3 Реализация с поддержкой вложенных сетей

Описанные модификации базовых классов с использованием MPI, как уже было оговорено выше, позволяют создавать автоматные сети, состоящие лишь из элементарных автоматов, не являющихся вложенными автоматными сетями. Вследствие этого при реализации сложной сети могут возникнуть неоправданно большие накладные расходы на зачастую ненужную передачу выходных данных между всеми процессами. Для реализации же полноценного параллелизма автоматных сетей с использованием MPI требуется внести небольшие изменения в предоставляемый описанными классами интерфейс. Как и в предыдущем случае,

```
 {
  // уничтожение автомата
  m_factory.destroy_fsm(m_fsmnum, m_pfsm.back());
 }
 // ...
};
```

Поскольку автомат в текущем процессе один, изменяется также код функции класса fsmnet_type, выполняющей работу сети на одном такте. Эта работа сокращается до соответствующего последовательного вызова трех функций автомата:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 void do_work(void)
 {
  // прочитать входные данные, выполнить действия
  m_pfsm.back()->put_input(m_shared.get_input(m_fsmnum));
  m_pfsm.back()->do_work();
  // записать выходные (запись должна быть отделена от чтения)
  m_shared.put_output(m_fsmnum, m_pfsm.back()->get_output());
 }
 // ...
};
```

Наконец, поскольку автоматы находятся в разных процессах, в функцию проверки выключения сети вносится операция глобальной редукции:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 bool is_off(void) const
 {
  // сеть в начальном состоянии, когда все автоматы в начальном
  int isoff = m_pfsm.back()->is_off();
  int allisoff;
  MPI_Allreduce(
   &isoff, &allisoff, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);
  return allisoff;
 }
 // ...
};
```

Все эти изменения коснулись класса fsmnet_type. Однако для взаимодействия автоматов сети между собой также потребуется изменение класса shared_area_type. Поскольку после записи выходных данных каждого автомата требуется их рассылка всем остальным процессам, в конец функции записи выходов автомата добавлен вызов MPI_Allgatherv, выполняющий сбор областей


```
 // количество выходных каналов
 virtual int number_output(void) const = 0;
 // передать автомату вектор входных данных
 virtual void put_input(const std::vector<data_type> &input) = 0;
 // выполнить такт работы автомата
 virtual void do_work(void) = 0;
 // получить от автомата вектор выходных данных
 virtual std::vector<data_type> get_output(void) const = 0;
 // проверка, находится ли автомат в начальном состоянии
 virtual bool is_off(void) const = 0;
};
```

Первые две функции возвращают информацию о количестве входных и выходных сигналов автомата. Следующие три функции выполняют соответственно работу по передаче автомату входных данных, выполнению автоматом внутренних действий и получению от автомата выходных данных. Наконец, последняя функция возвращает информацию о том, находится ли автомат в настоящий момент в выключенном (начальном/конечном) состоянии.

На основе абстрактного типа автомата создается тип элементарного конечного автомата (fsm_type). Функции этого класса определяют в общем виде работу элементарного конечного автомата за исключением конкретных действий. Помимо возврата основных характеристик и хранения входных и выходных значений автомата, этот класс инкапсулирует хранение и использование таблицы обработчиков состояний.

Обработчик состояния – функция-член класса конкретного автомата, который будет унаследован от класса fsm_type. Каждому состоянию, объявленному унаследованным классом, им же сопоставляется некий обработчик и добавляется в таблицу обработчиков функцией add_handler.

Во время каждого такта в зависимости от состояния вызывается тот или иной обработчик. Один обработчик может быть подключен более чем к одному состоянию. В большинстве литературы для выбора обработчиков рекомендуют использовать оператор switch или его аналоги в других языках [4, 13, 27]. Однако такая конструкция не всегда удобна при динамическом формировании автомата во время выполнения. В этой ситуации может быть использована альтернативная конструкция на основе

if – else if – else, однако время поиска нужного обработчика в таком случае будет расти линейно в зависимости от количества состояний, что при большом количестве состояний обусловит длительные задержки. Поэтому мы воспользовались стандартным шаблоном map, который реализуется двоичным деревом и потому обеспечивает время поиска, пропорциональное логарифму количества состояний.

Наконец, на основе абстрактного типа автомата объявляется тип сложного автомата – автоматной сети (fsmnet_type).

Конструктор и деструктор этого класса осуществляют соответственно создание и уничтожение внутренних автоматов средствами так называемой «фабрики сети». Объект класса фабрики сети, объявляемого вызывающим кодом на основе абстрактного класса factory_abstract_type, передается конструктору класса fsmnet_type и содержит всю необходимую информацию о создаваемой автоматной сети.

Класс fsmnet_type содержит вложенный класс shared_area_type, инкапсулирующий создание и использование общей области памяти автоматной сети для хранения промежуточных значений выходов автоматов. Создание области памяти сопровождается также созданием списков индексов для чтения входных данных и для записи выходных данных каждым автоматом. Все данные для работы с общей областью данных создаются в конструкторе класса shared_area_type на основе списка связей, переданных от фабрики сети.

Список связей формируется в фабрике сети вызывающим кодом с помощью вспомогательного класса linkstore_type, который введен для простоты описания добавления связи.

Два слова о фабрике сети. Этот класс введен по той причине, что автоматы в явном виде должны создаваться вызывающим кодом. В то же время список созданных автоматов не должен передаваться конструктору готовым, так как в целях удобства и унификации распараллеливания конструктор сам должен принимать решение о создании конкретных автоматов. Помимо этого, класс фабрики сети выполняет еще одну роль, о которой подробнее будет сказано чуть позже. Сейчас лишь оговоримся, что этот

памятью. Для краткости, мы будем приводить лишь те фрагменты кода, приведенного в приложении 3, которые подверглись изменению.

Для начала рассмотрим простейший способ реализации, требующий минимальных изменений вызывающего кода, но который, однако, будет работать только для случая отсутствия вложенных сетей, т.е. когда вся сеть состоит из элементарных автоматов.

Предположим для простоты, что количество процессов в группе коммуникатора MPI_COMM_WORLD соответствует количеству автоматов сети, и на каждый автомат приходится свой процесс. Тогда в каждом процессе создается объект сети, содержащий один автомат. В этой ситуации объекту сети в каждом процессе, прежде всего, необходимо знать номер конкретного автомата, им реализуемого. Этот номер хранится в переменной-члене класса fsmnet_type:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
private:
 // номер автомата текущей сети, реализуемого нашим процессом
 int m_fsmnum;
 // ...
};
```

Присвоение номера автомата производится путем получения ранга текущего процесса в коммуникаторе MPI_COMM_WORLD в теле конструктора fsmnet_type. Там же выполняется создание одного автомата с текущим номером. В деструкторе, соответственно, выполняется уничтожение одного автомата с известным номером:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
public:
 fsmnet_type(factory_abstract_type &factory):
  m_factory(factory), m_shared(m_factory.get_links().get())
 {
  MPI_Comm_rank(MPI_COMM_WORLD, &m_fsmnum);
  // создание автомата
  m_pfsm.push_back(m_factory.create_fsm(m_fsmnum));
 }

 ~fsmnet_type(void)
```

В приведенном примере на вход сети подается лишь два варианта векторов – некий начальный вектор, инициирующий работу сети, и вектор нулевых значений. После выполнения этого кода объект vouts содержит по порядку все векторы выходных значений, кроме вектора последнего такта. Вектор выходных значений последнего такта нас не интересует, поскольку после последнего такта сеть пребывает в выключенном состоянии, поэтому полезные сигналы, в т.ч. выходные, отсутствуют.

В последнем примере на каждом такте осуществляется передача входных параметров сети и изъятие выходных. В некоторых частных случаях это не является необходимым, поскольку входные параметры на протяжении всей работы сети не меняются или их изменение во время работы сети не учитывается, а выходные параметры могут отсутствовать либо нас не интересовать (к примеру, когда сеть выполняет какую-либо внутреннюю работу с инкапсулированной в автоматах передачей данных вовне, не заключающуюся в выдаче полезного результата на выход сети). В такой ситуации цикл можно было существенно упростить, исключив вызовы put_input и get_output из тела цикла:

```
fsmnet.put_input(vin);
do
 fsmnet.do_work();
while (!fsmnet.is_off());
```

В этом примере вектор входных значений помещается в сеть перед первым тактом и не меняется на протяжении всего цикла работы сети.

3.3.2 Простая реализация с использованием MPI

Приведенный в приложении 3 набор общих базовых классов сделан по возможности универсальным с точки зрения инкапсуляции создания и работы автоматов для того, чтобы использующий эти классы код подвергался минимальным изменениям при смене инструмента распараллеливания. Как видно, этот код был выполнен последовательным, после чего был распараллелен с помощью директив OpenMP. Теперь рассмотрим, как приведенный набор классов может быть распараллелен с использованием интерфейса MPI для выполнения в системах с распределенной

класс должен быть «легким», т.е. как таковое создание объекта этого класса не должно требовать больших затрат ресурсов.

Для создания автоматной сети в вызывающем коде должен быть объявлен набор классов автоматов, унаследованных от fsm_type, а также класс фабрики сети, унаследованный от factory_abstract_type. Во время выполнения вызывающим кодом должен быть создан объект фабрики сети, который должен существовать на протяжении всего жизненного цикла сети, после чего должен быть создан объект fsmnet_type с передачей конструктору созданного объекта фабрики сети. Объекты конкретных автоматов из объявленного набора классов должны создаваться и уничтожаться внутри функций фабрики сети create_fsm и destroy_fsm соответственно.

К примеру, объявление сети из двух автоматов может выглядеть следующим образом:

```
class fsm1_type: public fsm_type
{
public:
 enum {
  INPUT_0,
  // ... прочие объявления входов
  INPUT_NUMBER};
 enum {
  OUTPUT_0,
  // ... прочие объявления выходов
  OUTPUT_NUMBER};
 enum {
  STATE_0,
  // ... прочие объявления состояний
  };

private:
 state_type handle_off(state_type state)
 {
  if (/* ... */)
  {
   state = STATE_0;
   m_output[OUTPUT_0] = /* ... */;
   // ... заполнение оставшихся выходов
  }
  // ... обработка прочих условий
  return state;
 }
 // ... объявления прочих обработчиков

public:
```

```
 fsm1_type():
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&fsm1_type::handle_off));
  // ... добавление прочих обработчиков
 }
};

class fsm2_type: public fsm_type
{
 // ...
};

class fsmnet1_type: public fsmnet_type
{
public:
 enum {FSM_1, FSM_2, FSM_NUMBER};
 enum {
  INPUT_0,
  // ... прочие объявления входов
  INPUT_NUMBER};
 enum {
  OUTPUT_0,
  // ... прочие объявления выходов
  OUTPUT_NUMBER};

 class factory_type: public factory_abstract_type
 {
 public:
  int number_fsm(void) const { return FSM_NUMBER; }
  int number_input(void) const { return INPUT_NUMBER; }
  int number_output(void) const { return OUTPUT_NUMBER; }
  linkstore_type get_links(void) const
  {
   linkstore_type store;

   store.add(
    PSEUDOFSM_NETINPUT, INPUT_0,
    FSM_1, fsm1_type::INPUT_0);
   // ... прочие соединения со входами сети

   store.add(
    FSM_2, fsm2_type::OUTPUT_0,
    PSEUDOFSM_NETOUTPUT, OUTPUT_0);
   // ... прочие соединения с выходами сети

   store.add(
    FSM_1, fsm1_type::OUTPUT_0,
    FSM_2, fsm2_type::INPUT_0);
   // ... прочие межавтоматные соединения

   return store;
  }
```



```
  fsm_abstract_type *create_fsm(int i)
  {
   if (i == FSM_1)
    return new fsm1_type();
   // ... создание прочих автоматов
  }
  void destroy_fsm(int i, fsm_abstract_type *pfsm)
  {
   if (i == FSM_1)
    delete dynamic_cast<fsm1_type *>(pfsm);
   // ... уничтожение прочих автоматов
  }
 };

 fsmnet1_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Работа такой сети должна выполняться внешним циклом так, как будто она является обычным конечным автоматом. В частности, она может быть осуществлена следующим циклом:

```
// два варианта входных векторов
vector<fsm_type::data_type> vin(fsmnet1_type::INPUT_NUMBER);
vector<fsm_type::data_type> vinzero(fsmnet1_type::INPUT_NUMBER);
fill(vinzero.begin(), vinzero.end(), 0);
// ... инициализация начального вектора

// массив выходных векторов
vector<vector<fsm_type::data_type> > vouts;

// фабрика автоматной сети и сама сеть
fsmnet1_type::factory_type factory;
fsmnet1_type fsmnet(factory);

// цикл работы сети, в начальный момент выключена
bool isoff = true;
do
{
 // подать входные данные, если сеть выключена
 fsmnet.put_input(isoff ? vin : vinzero);
 // выполнить такт работы сети
 fsmnet.do_work();
 // проверить, выключена ли сеть
 isoff = fsmnet.is_off();
 // получить очередные выходные данные
 if (!isoff)
  vouts.push_back(fsmnet.get_output());
} while (!isoff);

// по завершении работы сети vouts.back() содержит
// значения выходов сети на предпоследнем такте
```