ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г. DOI 10.17587/issn.2220-3397 ISSN 2220-3397

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индекс по Объединенному каталогу "Пресса России" — **22765**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru/prin/rus E-mail: prin@novtex.ru
Журнал включен в систему Российского индекса научного цитирования и базу данных RSCI на платформе Web of Science.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

Vol. 10
N 4
2019

Published since September 2010 **DOI 10.17587/issn.2220-3397** **ISSN 2220-3397**

CONTENTS

Information about the jornal is available online at:
http://novtex.ru/prin/eng e-mail:prin@novtex.ru

УДК 004.023 DOI: 10.17587/prin.10.147-159

В. А. Галатенко, д-р физ.-мат. наук, зав. сектором, galat@niisi.ras.ru,
Н. И. Вьюкова, ст. науч. сотр., e-mail: niva@niisi.ras.ru,
К. А. Костюхин, канд. физ.-мат. наук, ст. науч. сотр., kost@niisi.ras.ru,
Федеральное государственное учреждение "Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук" (ФГУ ФНЦ НИИСИ РАН), Москва

Схемы программ как инструмент распараллеливания. Основные понятия*

Схемы программ как средство накопления и использования программистских знаний рассматриваются в настоящей работе в контексте параллельного программирования. Они могут служить основой тестов для оценки производительности параллельных архитектур и/или продуктивности программистов, разрабатывающих новые параллельные приложения или распараллеливающих унаследованный код. Схемы программ могут быть полезны для разработки и реализации аппаратуры, поддерживающей параллелизм, например, ПЛИС, сконфигурированные соответствующим образом. Настоящая статья является первой частью работы, представляющей результаты исследований, направленных на анализ особенностей описания схем программ как инструментария для их применения в различных аппаратно-программных системах. В ней рассматриваются основные понятия и определения, а также приводится обзор существующих архитектурных решений.

Ключевые слова: *схемы программ, параллельное программирование, распределенные системы, обзор*

Введение

Схемы программ, например, представленные в работе [1], являются средством накопления и многократного использования программистских знаний. Они полезны в первую очередь при разработке программного обеспечения, так как позволяют программисту мыслить крупными категориями и действовать в духе сборочного программирования, избавляя программиста от рутинных ошибок. Последнее особенно важно для разработки параллельных систем, характеризующихся повышенной сложностью.

На схемы программ можно смотреть и с другой стороны, выявляя в последовательных программах фрагменты, допускающие распараллеливание известными способами. Подобными схемами может оперировать компилятор, используя их для оптимизации.

Таким образом, применение схем программ как инструментального средства распараллеливания предполагает решение двух задач:

— накопление набора схем программ с достаточным для практического использования покрытием предметной области;

— разработка и реализация набора трансформаций, преобразующих схемы к виду, допускающему эффективную компиляцию для выбранных целевых конфигураций.

Настоящая статья является первой частью работы, представляющей результаты исследований, направленных на анализ особенностей описания схем программ как инструментария для их применения в различных аппаратно-программных системах. Рассмотрены основные понятия и определения, а также дан обзор существующих архитектурных решений.

Понятие схемы программы

Схема программы — это специальным образом описанный (откомментированный) и параметризованный фрагмент кода, который можно настраивать, используя для решения конкретных задач.

Формально схема программы определяется как следующая четверка:

- аннотация;
- параметры;
- спецификации;
- тело.

Аннотации содержат ключевые слова и служат для облегчения поиска нужной схемы. В нашей работе их обсуждать не будем.

Параметрами схемы могут служить любые языковые объекты — константы, переменные, типы, действия.

* Публикация выполнена в рамках государственного задания по проведению фундаментальных научных исследований по теме (проекту) "38. Проблемы создания глобальных и интегрированных информационно-телекоммуникационных систем и сетей, развитие технологий и стандартов GRID. Исследование и реализация программной платформы для перспективных многоядерных процессоров (0065-2019-0002)."

Спецификация схемы состоит из трех логических частей, которые можно обозначить следующим образом:

- условия применимости;
- внутренние ограничения;
- результирующие утверждения.

Условия применимости содержат ограничения на параметры схемы. Внутренние ограничения — это ограничения, налагаемые на параметры-действия, которые удобно формулировать внутри тела схемы. Наконец, результирующие утверждения описывают ситуацию после применения схемы.

Можно ввести понятие корректности схемы. Интуитивно оно очевидно, а именно если из условий применимости и внутренних ограничений в соответствии с семантикой схемы следуют результирующие утверждения, то схема корректна.

Условия применимости и результирующие утверждения задаются как совокупность логических выражений. Внутренние ограничения удобно специфицировать в рамках формализма Хоара.

Спецификации схемы могут содержать предикаты, относящиеся к конкретной предметной области. Предполагается, что эти предикаты известны инструментальной системе поддержки разработки. Без такой настройки на предметную область сформулировать содержательные спецификации было бы невозможно.

Схемы не привязаны к какому-либо языку программирования и могут задаваться по-разному. Например, их можно представлять себе как параметризованные шаблоны в C++.

Приведем пример схемы, вычисляющей минимальное целое число, большее некоторой нижней границы и удовлетворяющее заданному свойству (листинг 1). Пример записан на псевдоязыке.

```
схема Поиск минимального
  параметры
    n, i0, P
  где
  { Спецификация схемы }
    { Условия применимости }
      n: целое
      i0: константа целое
      P: процедура (целое) -> логическое
      существует n > i0: P(n)
  результаты
    { Результирующие утверждения }
      n = min {m | m > i0 & P(m)}
  это
  { Тело схемы }
    n := i0
    повторять
      n := n + 1
    до P(n)
конец схемы Поиск минимального
```

Листинг 1. Пример 1 схемы программы (данная схема не содержит внутренних ограничений)

Приведем еще один пример, в котором отыскивается какое-нибудь целое число, принадлежащее некоторому диапазону и удовлетворяющее заданному свойству (листинг 2).

```
схема Поиск подходящего
  параметры
    n, i0, i1, P
  где
  { Спецификация схемы }
    { Условия применимости }
      n: целое
      i0, i1: константа целое
      i0 <= i1
      P: процедура (целое) -> логическое
      существует n: P(n)
  результаты
    { Результирующие утверждения }
      i0 <= n <= i1 & P(n)
  это
  { Тело схемы }
    для всех i0 <= n <= i1 повторять
    до P(n)
конец схемы Поиск подходящего
```

Листинг 2. Пример 2 схемы программы

Между внешне схожими примерами 1 и 2 есть важное различие: цикл в примере 2 может быть распараллелен.

Схемы программ могут служить основой для расширения произвольного языка программирования вверх и для настройки его на конкретную предметную область. С помощью схем языковые конструкции повышенного уровня преобразуются к виду, допускающему эффективную компиляцию. С точки зрения преобразователя ситуация выглядит следующим образом.

Имеется формулировка задачи, которая состоит из фактов об исходных данных и требованиях на результат ее решения. Факты и требования имеют тот же вид, что и спецификация схемы, но, естественно, без формальных параметров. Чтобы показать, что схема решает задачу, нужно продемонстрировать существование такого означивания формальных параметров схемы программными объектами задачи, что

— из фактов об исходных данных задачи следуют условия применимости схемы;

— выполнены внутренние ограничения;

— из результирующих утверждений схемы следуют требования на результаты задачи.

Интуитивно это означает, что можно построить специализацию схемы и получить программу, решающую поставленную задачу. Нетрудно доказать, что если схема корректна и решает поставленную задачу, то в соответствии с семантикой схемы после ее подстановки из фактов об исходных данных будут следовать требования на результаты.

Далее в демонстрационных примерах будут использоваться различные языки программирования без строгого соблюдения описанной выше структуры схем.

Схемы для распараллеливания на уровне операционной системы

Операционная система, поддерживающая потоки управления, служит основой для распараллеливания прикладных программ. Стандартом POSIX предусмотрено несколько механизмов обслуживания подобных приложений. Один из них — переменные условия.

Пусть имеется некоторый предикат (условие), зависящий от значений переменных, разделяемых несколькими потоками управления. Совместное использование мьютексов, переменных условия и обслуживающих их функций позволяет организовать экономное ожидание состояния истинности этого предиката. Под экономным ожиданием авторы подразумевают минимальное расходование потоком управления таких ресурсов, как, например, память и процессорное время.

Общая схема применения переменных условия выглядит следующим образом. С разделяемыми переменными, фигурирующими в предикате, ассоциируется мьютекс, который необходимо захватить перед началом проверок. Затем поток управления входит в цикл вида, показанного на листинге 3.

```
while (! предикат) {
Ожидание на переменной условия с освобождением
мьютекса. После успешного завершения ожидания
поток вновь оказывается владельцем мьютекса.
}
```

Листинг 3. Типичный цикл ожидания на переменной условия

После нормального выхода из цикла проверяемое условие истинно, можно выполнить требуемые действия и освободить мьютекс.

Подобный цикл ожидания может оказаться бесконечным. Чтобы ограничить его по времени, можно воспользоваться схемой, приведенной на листинге 4.

```
rc = 0;
while (! predicate && rc == 0) {
   rc = pthread _ cond _ timedwait
   (&cond, &mutex, &ts);
}
```

Листинг 4. Типичный цикл ожидания на переменной условия с контролем по времени

Традиционной проблемой в многопоточной среде является управление индивидуальными данными потоков. Следующая программа иллюстрирует возможное решение этой проблемы. Здесь параметром схемы могли бы стать действия, выполняемые с индивидуальными данными (листинг 5).

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа запоминает в качестве индивидуальных данных */
/* потока управления время начала активных операций      */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>

static pthread _ key _ t data _ key;
static pthread _ once _ t key _ once = PTHREAD _ ONCE _ INIT;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Деструктор индивидуальных данных, в роли которых  */
/* выступает указатель на структуру типа timeval.    */
/* Поскольку она не содержит указателей,             */
/* достаточно освободить занимаемую ею память */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
static void data _ destructor (void *p) {
   free (p);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Функция создания ключа индивидуальных данных, */
/* ассоциирующая с ним деструктор, освобождающий память  */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
static void create _ data _ key (void) {
   (void) pthread _ key _ create (&data _ key, data _ destructor);
}
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Функция инициализации индивидуальных данных. */
/* Запрашивает астрономическое время             */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void *start _ func (void) {
   struct timeval *tmvl _ ptr;

/* Запомним астрономическое время начала операций потока управления */
   if ((tmvl _ ptr =
        (struct timeval *) malloc (sizeof (struct timeval))) == NULL) {
   return (NULL);
}
(void) gettimeofday (tmvl _ ptr, NULL);

   /* Создадим ключ индивидуальных данных,     */
   /* перепоручив вызов pthread _ key _ create() */
   /* функции pthread _ once()                  */

   (void) pthread _ once (&key _ once, create _ data _ key);

   (void) pthread _ setspecific (data _ key, tmvl _ ptr);

   return (tmvl _ ptr);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Функция main() вызывает функцию инициализации */
/* и запрашивает индивидуальные данные           */
/* потока управления */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
int main (void) {
   struct timeval *tmvl _ ptr;
   if (start _ func () = = NULL) {
     return (1);
   }

   if ((tmvl _ ptr =
        (struct timeval *) pthread _ getspecific (data _ key))!= NULL) {
     printf ("Время начала операций потока управления: %ld сек, %ld мсек\n",
        tmvl _ ptr->tv _ sec, tmvl _ ptr->tv _ usec);
   } else {
     printf ("Отсутствуют индивидуальные данные потока управления.\n");
     printf ("Время начала операций неизвестно\n");
     return (2);
   }
   return 0;
}
```

Листинг 5. Пример программы, формирующей и опрашивающей индивидуальные данные потоков управления

Одним из средств синхронизации в многопотоковой среде являются барьеры. Типовая схема работы с ними проиллюстрирована листингом 6.

```
if ((status = pthread _ barrier _ wait (&barrier)) ==
     PTHREAD _ BARRIER _ SERIAL _ THREAD) {
   /* Выделенные (обычно — объединительные) действия. */
   /* Выполняются каким-то одним потоком управления   */
} else {
   /* Эта часть выполняется всеми прочими потоками управления */
   if (status! = 0) {
```

```
    /* Обработка ошибочной ситуации */
  } else {
    /* Нормальное "невыделенное" завершение ожидания на барьере */
  }
}

/* Повторная синхронизация — */
/* ожидание завершения выделенных действий */
status = pthread_barrier_wait (&barrier);

/* Продолжение параллельной работы */
  . . .
```

Листинг 6. Типовая схема применения функции `pthread_barrier_wait()`

Стандартом POSIX предусмотрено создание альтернативного стека для функции обработки сигналов реального времени. На листинге 7 показана типовая схема определения альтернативного стека.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
  . . .
stack_t sighstk;
  . . .
if ((sighstk.ss_sp = malloc (SIGSTKSZ)) == NULL) {
  perror ("malloc (SIGSTKSZ)");
  /* Аварийное завершение */
}
sighstk.ss_size = SIGSTKSZ;
sighstk.ss_flags = 0;
if (sigaltstack (&sighstk, (stack_t *) NULL) != 0) {
perror ("SIGALTSTACK");
    . . .
}
  . . .
```

Листинг 7. Типовая схема определения альтернативного стека

Схемы программ могут использоваться не только в положительном, но и в отрицательном смысле, т. е. для выявления типичных ошибок. Очевидно, процессу может выделяться сколь угодно малая доля процессорного времени, так что соотношение между виртуальным и реальным временем оказывается недетерминированным. Если, например, расположить вызов `setjmp()` после взведения таймера (листинг 8), то таймер реального времени может сработать до инициализации буфера `buf_env`, после чего нелокальный переход из функции обработки сигнала приведет к полному хаосу.

```
if (setitimer (ITIMER_REAL, &itvl, NULL) < 0) {
  . . .
}

  . . .
(void) setjmp (buf_env);
```

Листинг 8. Пример некорректной последовательности взведения таймера и инициализации структуры для нелокального перехода

Другие примеры схем программ для распараллеливания на уровне операционной системы можно найти в работах [2, 3].

Схемы для распараллеливания на уровне распределенных, разнородных систем

Будем рассматривать многомашинные комплексы, состоящие из многоядерных систем с неоднородным доступом к памяти и со специализированными сопроцессорами.

Межмашинное взаимодействие может осуществляться с использованием спецификаций MPI. Многоядерность может поддерживаться интерфейсом Pthreads и/или OpenMP. Для описания разнородных конфигураций служит модель OpenCL. Соответственно, возникает два направления масштабирования: вверх, когда увеличивается число ядер; вширь, когда возрастает число взаимодействующих машин.

Прежде чем применять схемы программ, разработчик должен:

— разбить исходную проблему на задачи, которые могут выполняться параллельно;

— определить, какие данные локальны для каждой задачи, а какие разделяются между задачами;

— выявить зависимости по порядку выполнения и по данным, существующие между задачами.

Вероятно, первой фундаментальной работой, посвященной применению схем программ для распараллеливания (*parallel patterns*), стала книга [4]. В ней рассмотрены три вида параллелизма:

- на уровне задач;
- на уровне данных;
- конвейеризация циклов.

Параллелизм на уровне задач существует, когда моделируется совокупность слабо связанных сущностей.

Параллелизм на уровне данных характерен для научных вычислений, например для сеточных методов.

Конвейеризация циклов полезна, когда цикл по результатам профилирования оказался узким местом.

В работе сделан акцент на системы с массовым параллелизмом и использование интерфейсов MPI и OpenMP. В дальнейшем изложении ограничимся примерами для OpenMP.

В качестве первого примера рассмотрим схему "неограниченная параллельность". Условие ее применимости состоит в том, что решение всей задачи распараллеливания в целом получается как комбинация решений независимых частных проблем (листинг 9).

```
solution (P) =
   f (subsolution (P, 0),
      subsolution (P, 1), ...,
      subsolution (P, N-1))

subsolution (P, i) и subsolution (P, j)
независимы для разных i и j.
```

Листинг 9. Условие применимости схемы "неограниченная параллельность"

Тело схемы "неограниченная параллельность" показано на листинге 10.

```
Problem P;
Solution subsolutions[N];
Solution solution;

for (i = 0; i < N; i++) {
   subsolutions [i] = compute _ subsolution (P, i);
}
solution = compute _ f(subsolutions);
```

Листинг 10. Тело схемы "Неограниченная параллельность"

Если задачи `compute _ subsolution(P, i)` независимы, итерации цикла могут выполняться параллельно (см. также листинги 1 и 2). Добиться этого можно разными способами. Например, сложение векторов может быть оформлено так, как показано на листинге 11.

```
!$OMP PARALLEL DO
      DO I = 1, N
        C(I) = A(I) + B(I)
        ENDDO
!$OMP END PARALLEL DO
```

Листинг 11. Конкретизация схемы "неограниченная параллельность" для сложения векторов

Здесь использована директива OpenMP. Отметим, что функция `compute _ f(subsolutions)` в данном случае оказалась пустой.

Параллелизм на уровне независимых задач может поддерживаться в рамках модели "мастер/работник". Работники берут задачи из общего хранилища (общей очереди задач), выполняют их и т. д. При этом автоматически обеспечивается балансировка нагрузки.

Реализация модели "мастер/работник" состоит из двух логических частей (листинги 12 и 13). Процесс `master` создает очередь задач, представляя каждую задачу целым числом. Затем посредством шаблона `ForkJoin` он создает процессы или потоки управления `worker` и ждет их завершения, после чего использует полученные результаты.

```
#define Ntasks 500            /* Число задач */
#define Nworkers 5            /* Число работников */

SharedQueue task _ queue;           /* Очередь задач */
Results Global _ results[Ntasks];   /* Массив для хранения результатов */

void master() {
   void Worker();

   // Создать и инициализировать разделяемые структуры данных
   task _ queue = new SharedQueue();

   for (int i = 0; i < N; i++)
     enqueue(&task _ queue, i);
```

```
   // Создать потоки управления для работников, выполняющие функцию Worker()
   ForkJoin (Nworkers, Worker);

   Consume _ the _ results (Ntasks);
   }
```

Листинг 12. Процесс `master` из модели "мастер/работник"

```
void Worker() {
   int i;
   Result res;

   While (!empty(task _ queue) {
     i = dequeue(task _ queue);
     res = do _ lots _ of _ work(i);
     Global _ results[i] = res;
   }
}
```

Листинг 13. Процесс `worker` из модели "мастер/работник"

Отметим, что безопасный доступ к ключевым общим переменным (очереди задач) обеспечивается схемами уровня операционной системы.

Класс схем "выделяемая параллельность" характеризуется тем, что их можно свести к неограниченной параллельности тиражированием разделяемых глобальных данных перед началом параллельной обработки и редукцией локальных результатов к глобальному результату после завершения обработки. Это возможно, если каждое глобальное данное модифицируется только одним процессом (потоком управления), а остальные процессы используют начальное состояние этого данного только на чтение. Пример численного интегрирования методом трапеций иллюстрирует возможную реализацию подобных схем на основе OpenMP (листинг 14).

```
#include <stdio.h>
#include <omp.h>
#define NUM _ THREADS 2
static long num _ steps = 100000;
double step;

void main () {
   int i;
   double x, pi, sum = 0.0;

   step = 1.0/(double) num _ steps;

   omp _ set _ num _ threads (NUM _ THREADS);

   #pragma omp parallel for reduction(+:sum)
   private(x) for (i = 1; i<= num _ steps; i++) {
     x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf (" pi is %f \n ", pi);
}
```

Листинг 14. Пример реализации на основе OpenMP схемы с выделяемой параллельностью

Более общей является ситуация, когда данные можно разбить на блоки, модифицируемые параллельно, однако модификация каждого блока при этом требует данные из других блоков. Соответствующие схемы программ полезны, например, при реализации сеточных вычислительных методов.

Реализацию подобных схем можно выполнить на основе OpenMP (листинг 15).

```
      real uk(1:NX), ukp1(1:NX)
      dx = 1.0/NX
      dt = 0.5*dx*dx
C-----инициализация uk, ukp1 опущена
      do k=1,NSTEPS
C$OMP     PARALLEL DO SCHEDULE(STATIC)
          do i=2,NX-1
            ukp1(i)=uk(i) + (dt/(dx*dx))*
             (uk(i+1)-2*uk(i) +uk(i-1))
          enddo
C$OMP     END PARALLEL DO
C$OMP     PARALLEL DO SCHEDULE(STATIC)
          do i=2,NX-1
            uk(i)=ukp1(i)
          enddo
C$OMP     END PARALLEL DO
          print _ step (k, uk)
      enddo
      end
```

Листинг 15. Пример реализации на основе OpenMP схемы с разбиением данных на параллельно обрабатываемые блоки

Поскольку OpenMP поддерживает среду с разделяемой памятью, нет необходимости явным образом делить на блоки и распределять два ключевых массива (`uk` и `ukp1`). Вычисление новых значений неявным образом распараллеливается двумя директивами OpenMP, которые также неявно обеспечивают требуемую синхронизацию (все потоки управления должны завершить изменение `ukp1` до начала изменения `uk` и наоборот). Конструкция `SCHEDULE(STATIC)` распределяет итерации параллельных циклов по доступным потокам управления.

Работа [5] сфокусирована на параллелизме по данным и оптимизации циклов. В ней на примере системы Spartan рассмотрен вопрос локальности данных применительно к распределенным массивам, приближение данных к вычислениям, автоматическое разбиение n-мерных массивов и циклов на блоки, максимизирующие локальность.

В основе системы Spartan лежат пять высокоуровневых параллельных операторов: `map`, `fold`, `filter`, `scan` и `join _ update`. С их помощью выражаются

типичные параллельные схемы большинства программ обработки массивов. Такие схемы используются для реализации большого числа встроенных и пользовательских функций.

Отметим, что перечисленные операторы являются исключительно функциональными: входные массивы никогда не изменяются, результаты записываются в новые места. Операторы параметризуются пользовательскими функциями, которые должны быть детерминированными и свободными от побочных эффектов.

Определяются операторы следующим образом.

- `D = map (f _ map, S1, S2,...)` применяет функцию `f _ map` параллельно по блокам к входным массивам `S1, S2,...` и генерирует выходной массив `D` той же формы.
- `D = filter (f _ pred, S)` создает представление массива `S`, из которого исключены элементы, не удовлетворяющие предикату `f _ pred`. Вместо `f _ pred` может использоваться булев массив. Поскольку `filter` создает представление без копирования реальных данных, накладные расходы на коммуникации отсутствуют.
- `D = fold (f _ accum, S, axis)` агрегирует входной массив `S` вдоль измерения axis с помощью коммутативной и ассоциативной функции `f _ accum`. Естественно, свертки выполняются параллельно.
- `D = scan (f _ accum, S, axis)` вычисляет накапливаемые агрегаты вдоль измерения `axis` массива `S`, используя функцию `f _ accum`. В отличие от `fold`, выходной массив `D` имеет ту же форму, что и `S`.
- `D = join _ update (f _ join, f _ accum, S1, S2, ..., axis1, axis2,..., output _ shape)`. Это более сложный оператор. Каждый входной массив `Si` трактуется как группа блоков вдоль `axisi`. Все входные массивы должны иметь одинаковое число блоков вдоль выбранных осей.

Spartan объединяет соответственные блоки из разных групп и параллельно применяет `f _ join`. Функция `f _ join` генерирует некоторые изменения, которые записываются в заданные места выходного массива `D`. Множество работников, выполняющих `f _ join`, могут параллельно изменять одно и то же место в `D`. Подобные конфликты автоматически разрешаются применением функции `f _ accum`. Особым образом в `join _ update` рассматривается случай, когда входной массив `Si` имеет ось `i = -1`. В этом случае весь массив `Si` объединяется с каждым блоком других входных массивов. Приведем несколько примеров использования высокоуровневых операций Spartan.

Листинг 16 иллюстрирует операцию `map`. Функция `numpy.add` складывает элементы массивов.

```
import numpy
import spartan
# Параллельная реализация в Spartan
# поэлементного сложения массивов
def add (a, b):
   return spartan.map (a, b, f _ map = numpy.add)
```

Листинг 16. Пример использования высокоуровневой операции `map`

На листинге 17 показана возможная реализация умножения матриц с использованием высокоуровневой операции `join _ update`. Первый сомножитель разбивается на блоки по столбцам, второй — по строкам. Пользовательская функция `dot _ udf` вызывается параллельно для этих блоков. Все изменения агрегируются с использованием сложения с накоплением, чтобы получить окончательный результат. Особый случай `join _ update (axisi = -1)` иллюстрируется в `else`-части листинга 17.

```
import numpy
import spartan

# Определяемая пользователем функция f _ join
def dot _ udf (input _ tiles):
   output _ loc = spartan.location (0,0)
   output _ data = numpy.dot (input _ tiles [0], input _ tiles [1])
   return output _ loc, output _ data

# Параллельная реализация в Spartan
# умножения матриц
def dot (a, b):
   if a.shape [0] < = a.shape [1]:
     return spartan.join _ update (S = (a, b), axes = (1, 0), f _ join = dot _ udf,
                                   shape =..., f _ accum = numpy.add)
   else:
     return spartan.join _ update (S = (a, b), axes = (0, -1),...)
```

Листинг 17. Пример использования высокоуровневой операции `join _ update`

В дополнение к пяти высокоуровневым операциям Spartan предоставляет несколько примитивов для создания распределенных массивов или представлений массивов.

- `D = newarray (shape, init _ method)` создает распределенный массив заданной формы. Массив может быть инициализирован разными способами: загрузкой данных из внешней памяти, заполнением нулями или случайными числами и т. д.
- `D = slice (S, region)` создает представление над заданной областью в массиве `S`. Дескриптор области задает начало и конец выделяемой области вдоль каждого измерения.
- `D = swapaxis (S, axis1, axis2)` создает представление массива `S` путем смены местами `axis1` и `axis2`. Обычно используется для транспонирования матриц.

Выходное представление `D` имеет отличное от `S` разбиение на блоки: разбиение вдоль столбцов, например, может быть заменено разбиением вдоль строк.

Для оптимизации пересылок данных при разбиении массивов на блоки используются жадные алгоритмы. Они дают не всегда оптимальный, но, как показано в работе [5], практически приемлемый результат.

Систему Spartan можно рассматривать как пример сочетания высокоуровневых схем, которые снижают затраты на разработку и сопровождение параллельных программ, с автоматическим обеспечением приемлемой эффективности.

Система Spartan ориентирована на высокопроизводительные вычисления, оперирующие с массивами. Целевая предметная область, о чем свидетельствует работа [6], это аналитика больших данных с использованием систем со значительным объемом внутренней памяти и неоднородным доступом к ней.

Основной предмет рассмотрения в работе [6] — язык распределенных мультициклов (Distributed MultiLoop Language, DMLL). В этой работе представлены высокоуровневые примитивы и набор правил переписывания, позволяющих эффективно отображать эти примитивы на различные целевые конфигурации. Язык DMLL поддерживает параллелизм по данным за счет неявно параллельных высокоуровневых шаблонов, а также вложенный параллелизм, т. е. вложенные параллельные конструкции с поддержкой параллелизма на всех уровнях.

Мультицикл — это абстракция цикла, поддерживающая слияние нескольких циклических шаблонов. Мультицикл представляет собой одномерный проход по фиксированному целочисленному диапазону, причем на каждой итерации может порождаться несколько значений. Каждый мультицикл содержит набор генераторов, отражающих высокоуровневую структуру тела цикла (параллельные шаблоны) и аккумулирующих результаты цикла. После завершения цикла генераторы возвращают результаты в программное окружение. Различают конвейерное и горизонтальное слияние циклов. Последнее позволяет за одну итерацию порождать несколько не связанных между собой результатов, т. е. иметь в теле цикла несколько генераторов.

Набор генераторов DMLL включает:

- `collect` — аккумулирует все сгенерированные значения и возвращает коллекцию как результат цикла, позволяя реализовать классические операции `map`, `zipWith`, `filter` и `flatMap`;
- `reduce` – выполняет редукцию "на лету" с использованием данной ассоциативной операции;
- `bucket-collect` — собирает значения в блоки, индексируемые ключами; если определена только функция, возвращающая ключ, получается `groupBy`;
- `bucket-reduce` — редуцирует значения в блоках.

Формально перечисленные генераторы определяются так, как показано на листинге 18.

```
G:: = Collect (s) (c) (f)              : Coll [V]
    | Reduce (s) (c) (f) (r)           : V
    | BucketCollect (s) (c) (k) (f)    : Coll [Coll [V]]
    | BucketReduce (s) (c) (k) (f) (r) : Coll [V]
c: индекс => Boolean условие
k: индекс => K   функция, возвращающая ключ
f: индекс => V   функция, возвращающая значение
r: (V, V) => V   редукция
s: Integer       размер цикла
```

Листинг 18. DMLL-генераторы

На листинге 19 показана возможная последовательная реализация генератора `collect`, рассчитанная на универсальные процессоры. Для графических процессоров можно первоначально вычислить условие для всех индексов, зарезервировать выходной буфер надлежащего размера, а затем еще одним циклом записать результаты в нужные места. Это подтверждает возможность оптимизации высокоуровневых схем с учетом целевой платформы.

```
[[Collect (s) (c) (f): Coll [V]]] =
  val out = new Coll [V]
  for (i <- 0 until s) {
    if (c (i)) out + = f (i)
  }
```

Листинг 19. Возможная последовательная реализация генератора `collect`, рассчитанная на универсальные процессоры

Правила переписывания позволяют выразить конвейерное слияние циклов (листинг 20, здесь `G` — произвольный генератор). Это — обобщение конвейеров `map-reduce`, `filter-groupBy` и т. п.

```
C = Collect (s)(c1)(f1)
G(C)(c2)(i =>
k(C(i)))(i =>
f2(C(i)))(r)
```

```
-> G(s)(c1 & c2)(k (f1))(f2(f1))(r)
```

Листинг 20. Правило переписывания для конвейерного слияния циклов

Ряд правил переписывания обслуживают вложенный параллелизм. На листинге 21 приведен пример подобного правила, применимого, когда имеется `BucketCollect`, результаты которого использует `Collect` с вложенным `Reduce` для каждого блока. Смысл правила в том, чтобы обработать данные за один проход, редуцируя величины при их распределении по блокам.

```
(GROUPBY-REDUCE)

A = BucketCollect(s)(c)(k)(f1)      -> H = BucketReduce
Collect (A ( _ )) (i =>                    (s)(c)(k)(f2(f1))(r)
Reduce (A (i)) ( _ ) (f2) (r))        Collect (H)( _ ) (i => H(i))
```

Листинг 21. Одно из правил переписывания (GROUPBY-REDUCE), обслуживающих вложенный параллелизм (подчеркивание обозначает тождественно истинное условие)

В работе [6] приведены результаты сравнения производительности (C++)-кода, оптимизированного вручную, с результатом генерации (C++)-кода из DMLL и последующей компиляцией посредством gcc 4.8 (уровень оптимизации 3). Автоматически сгенерированный код проиграл ручному не более 25 %, что является вполне удовлетворительным результатом.

Схемы для распараллеливания на уровне многоядерных систем

Общий принцип решения сложных задач формулируется как "разделяй и властвуй" (Divide-and-Conquer, DaC). В соответствии с этим принципом такое решение подразделяется на три стадии:

- разделение задачи на подзадачи;
- решение подзадач;
- получение решения задачи из решений подзадач.

Этот принцип обычно применяется рекурсивно, пока подзадача не станет достаточно простой для прямолинейного решения.

В работе [7] сделана попытка избавить разработчиков от ручного распараллеливания и предоставить им возможность сосредоточиться на содержательной стороне задач и привычном последовательном программировании. Отправной точкой для этой работы послужила работа [4]. В работе [7] предложена одна, но универсальная схема, реализующая принцип "разделяй и властвуй". Эта схема оформлена в виде параметризованного шаблона C++.

Интерфейс шаблона (по сути — условие применимости) показан на листинге 22.

```
// Типы функций
// "Разделяй"
using divide _ f _ t = std::function <void (const ProblemType&, std::vector <ProblemType>&)>;
// Комбинируй общее решение из решений подзадач
using combine _ f _ t = std::function <void (std::vector <ResultType>&, ResultType&)>;
// Прямолинейное решение ("властвуй")
using base _ f _ t = std::function <void (const ProblemType&, ResultType&)>;
// Условие: разбивать на подзадачи или решать прямолинейно
using cond _ f _ t = std::function <bool (const ProblemType&)>;

// DAC-конструктор шаблона
template <typename ProblemType, typename ResultType>
DAC (const divide _ f _ t& divide,
     const combine _ f _ t& combine,
     const base _ f _ t& base, const cond _ f _ t& cond,
     const ProblemType& p, ResultType& res,
     int par _ degree = available _ cores ())
```

Листинг 22. Интерфейс шаблона DAC "разделяй и властвуй"

Элемент `par _ degree` контролирует степень распараллеливания при применении шаблона. По умолчанию значение устанавливается равным числу имеющихся ядер.

Алгоритмическая реализация принципа "разделяй и властвуй" представлена на листинге 23.

```
void DACAlgo(const ProblemType &p, ResultType &ret) {
   if (!cond (&p)) { // Небазовый случай
     // "Разделяй"
     std::vector<ProblemType> ps;
     divide (p, ps);
     std::vector <ResultType> res (ps.size ());
```

```
    // "Властвуй" — рекурсивная фаза
    for(size_t i = 0; i < ps.size (); i + +)
      DACAlgo (ps [i], res [i]);
    // Комбинируй
    combine (res, ret);
    return;
  }
  seq (p, ret); // Базовый случай
}
```

Листинг 23. DAC-алгоритм

Если иметь в виду реализацию приведенного алгоритма средствами OpenMP, то рекурсивные вызовы `DACAlgo` нужно определять как задачи посредством соответствующего оператора `pragma`, а перед комбинированием результатов необходимо аналогичным образом обеспечить синхронизацию.

Предметом рассмотрения в работе [8] является конвейерное распараллеливание для многоядерных архитектур с разделяемой памятью. В основе этой работы лежит мультимасштабная гибридная программная модель (*Multiscale Hybrid Programming Model*, MHPM), которая позволяет легко выразить как последовательное выполнение, так и различные виды параллелизма, включая параллелизм по задачам, по данным и по времени (этот последний называется также конвейерным), с разными уровнями детализации. Модель MHPM реализована в виде (C++)-каркаса, называемого XPU.

В XPU задача — это по сути любой фрагмент кода, который можно вызвать и выполнить, а именно — функция, лямбда-выражение или метод класса. Задача может потреблять и/или поставлять данные. Пример определения задачи с использованием лямбда-выражения представлен на листинге 24. Этот и последующие примеры относятся к области обработки изображений.

```
image_t * stream;
xpu::task blur_t ([] (int i, image_t * video, int w, int h)
   { blur (video [i], w, h) }, // вызов исходной функции
          stream, 640, 480); // аргументы задачи
```

Листинг 24. Определение задачи, использующее лямбда-выражение

После того как задачи определены, их можно сделать стадиями в конвейере. При создании конвейера задачи передаются как аргументы в том порядке, в котором они должны выполняться (листинг 25). Первый аргумент конструктора конвейера `(n)` задает число элементов, подлежащих обработке. Когда это сделано, конвейер останавливается. Для бесконечной работы следует задать 0. Конвейер запускается вызовом метода `run ()`.

```
task greyscale_t (greyscl_stg, 0, stream,
width, height);
task blur_t (blur_stg, 0, stream, width,
height);
task threshold_t (threshold_stg, 0, stream,
width, height);
task multiply_t (mul_stg, 0, video, width,
height);
// Создать конвейер
task_group * process_image = pipeline (n,
                                       &greyscale_t,
                                       &threshold_t,
                                       &blur_t,
                                       &multiply_t);
process_image -> run (); // Запустить конвейер
```

Листинг 25. Пример создания конвейера из четырех стадий

С точки зрения операционной системы задачи представляются функционирующими параллельно потоками управления с необходимой синхронизацией между ними как стадиями конвейера. В XPU конвейер реализуется на основе отношения поставщик/потребитель между стадиями. Каждая стадия, кроме первой, ждет `(wait)` данные от предшественника, обрабатывает их, затем уведомляет `(notify)` преемника о готовности передать ему данные.

Направленные асинхронные коммуникации между параллельными потоками управления обслуживают событийные `(event)` объекты. Общий событийный объект используется для установления взаимодействия между парой потоков управления. У этого объекта два метода — `wait` и `notify`. Метод `wait` блокирует вызвавший его поток до поступления уведомления о готовности данных. Метод `notify` посылает уведомление потоку, ждущему это событие. Уведомления выстраиваются в очередь, так что при отсутствии ждущего они не пропадают, а при непустой очереди `wait` не блокирует поток, а просто берет первый элемент из очереди.

В XPU событийные объекты реализованы с использованием переменных условия. Очередь является потоково безопасной. Эффективность функционирования конвейера зависит от выбранной дисциплины планирования потоков управления (задач). В XPU предусмотрены три дисциплины: стандартная дисциплина операционной системы, балансировка нагрузки и локальность кэша.

Достоинство первой дисциплины — простота реализации. Для каждой стадии конвейера создается свой поток управления, операционная система распределяет задачи по процессорам и осуществляет планирование. Однако для конвейеров с большим числом стадий потоков может оказаться много, а нагрузка на разные процессоры может быть плохо сбалансированной.

Для балансировки нагрузки в XPU используется описанная выше модель "мастер/работник", т. е. создается постоянный пул потоков управления, которые берут работы из общей очереди. В каждую работу встроен событийный объект, показывающий, какой стадии конвейера она предназначена.

Дисциплина планирования для обеспечения эффективного использования кэша с каждым постоянным работником ассоциирует свою очередь работ. Работы, относящиеся к одному входному элементу

данных, на всех стадиях конвейера выполняются на одном процессоре. Недостатком этой дисциплины является возможная несбалансированность нагрузки.

В работе [8] представлены результаты сравнения XPU с другим средством распараллеливания — Threading Building Blocks (TBB) корпорации Intel [9]. Реализация XPU выполняется примерно на 20 % быстрее и требует существенно меньше специфически параллельного кода, т. е. способствует более высокой производительности труда программистов.

В статье [10], предшествующей работе [6], авторы описали эффективное отображение вложенного параллелизма на архитектуры с графическими процессорами.

На листинге 26 приведены два примера вложенных распараллеливаемых схем, оформленных как высокоуровневые операции. `sumCols` складывает элементы в столбцах матрицы, `sumRows` — в строках. Здесь внешняя операция — `map`, внутренняя — `reduce`.

```
// m: матрица размера [R, C]
sumCols = m mapCols { c => c reduce { (a,b) => a + b } }
sumRows = m mapRows { r => r reduce { (a,b) => a + b } }
```

Листинг 26. Пример вложенного параллелизма

Существует несколько стратегий отображения вложенных распараллеливаемых схем на конфигурации с графическими процессорами. Простейшая одномерная стратегия распараллеливает только верхний уровень, выделяя GPU-поток каждой итерации внешнего цикла Map. Двумерная стратегия может выделять каждой итерации внешнего цикла блок потоков, распараллеливая внутреннюю схему по потокам в блоке. Близкая стратегия использует вместо блоков потоков варпы.

Как показано в работе [10], ни одна из этих стратегий не является универсальной, даже если иметь в виду варьирование только формы матрицы из листинга 26. Может оказаться слишком мало потоков, слишком много блоков потоков, может также сказаться нелокальность доступа к памяти смежными потоками. Эффективность применения одной и той же стратегии может различаться в 50 раз.

Чтобы справиться с этой проблемой, в работе [10] предложен аналитический каркас, в котором вложенные схемы (от внешнего уровня 0 вглубь) отображаются в логически многомерное пространство, а размеры блоков и степень распараллеливания параметризуются по каждому измерению. Максимизируются локальность доступа к памяти и использование вычислительных ресурсов. Учитываются также жесткие и мягкие ограничения, специфичные для графических процессоров.

Отображения поддержаны компиляторными оптимизациями, использующими программно-управляемую разделяемую память и вынесение из циклов запросов на выделение памяти. В итоге удается автоматически получать программы, уступающие ручному программированию по эффективности результата не более чем на четверть и превосходящие на порядок фиксированные стратегии.

На листинге 27 приведен CUDA-код, сгенерированный для `sumRows`. Внутренняя схема редукции использует разделяемую память для комбинирования данных из потоков. Это общее место при программировании варпов, которое для краткости изложения опущено.

```
// Уровень 0: [Размерность Y, размер 64, максимальное распараллеливание]
// Уровень 1: [Размерность X, размер 32, отсутствие распараллеливания]
__ global __ kernel (double *m, int cols, double *out) {
   int y = threadIdx.y + blockIdx.y *
   blockDim.y;
   __ shared __ double smem [64] [32];
   double local _ sum = 0.0;

   for (int cidx = threadIdx.x; cidx < cols;
   cidx += 32)
     local _ sum += m [y*cols + cidx];
   smem [threadIdx.y] [threadIdx.x] = local _
   sum;
   __ syncthreads ();

   /* Редукция 32 значений в smem [threadIdx.y] [*] */
   if (threadIdx.x = = 0) out [y] =
   smem [threadIdx.y] [0];
}
```

Листинг 27. Сгенерированный CUDA-код для `sumRows`

Работа [10] подтверждает универсальный характер методов распараллеливания, предложенных в работе [6].

Заключение

Схемы программ — универсальное средство накопления и использования программистских знаний, особенно важное и полезное в сложных предметных областях, каковым является параллельное программирование. В настоящей работе даны основные определения и приведен обзор существующих решений в области применения схем программ в различных аппаратно-программных системах.

В части 2 работы будут рассмотрены основные механизмы применения схем программ при работе с параллельными системами.

Список литературы

1. **Бетелин В. Б., Галатенко В. А.** ЭСКОРТ — инструментальная среда программирования // Юбилейный сборник трудов институтов Отделения информатики РАН. 1993. Т. 2. С. 77—97.
2. **Галатенко В. А.** Программирование в стандарте POSIX / Под ред. академика РАН В. Б. Бетелина. М.: ИНТУИТ.РУ, 2004. 560 с.
3. **Галатенко В. А.** Программирование в стандарте POSIX. Часть 2 / Под ред. академика РАН В. Б. Бетелина. М.: ИНТУИТ.РУ, 2004. 424 с.
4. **Mattson T., Sanders B., Massingill B.** Patterns for parallel programming. Addison-Wesley Professional, 2004. 384 p.

5. **Chien-Chin Huang, Qi Chen, Zhaoguo Wang** et al. Spartan: A Distributed Array Framework with Smart Tiling // Proc. of the 2015 USENIX Annual Technical Conference (USENIC ATC 15), Santa Clara, CA, USA, 2015. P. 1—15.
6. **Brown K. J., Lee H. J., Rompf T.** et al. Have Abstraction and Eat Performance. Optimized Heterogeneous Computing with Parallel Patterns // Proc. of CGO'16, Barcelona, Spain, March 12—18, 2016. P. 194—205.
7. **Danelutto M., De Matteis T., Mencagli G., Torquati M.** A Divide-and-Conquer Parallel Pattern Implementation for Multicores // Proc. of SEPS'16, Amsterdam, Netherlands, November 1, 2016. P. 10—19.
8. **Khammassi N., Le Lann J.-C.** A High-Level Programming Model to Ease Pipeline Parallelism Expression On Shared Memory Multicore Architectures // Proc. of HPC'14, Tampa, Florida, April 13—16, 2014. Article 9.
9. **Intel** Corporation, Threading Building Blocks, TBB. URL: http://www.threadingbuildingblocks.org (дата обращения 23.01.2019).
10. **Lee H. J., Brown K. J., Sujeeth A. K.** et al. Locality-Aware Mapping of Nested Parallel Patterns on GPUs // Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, United Kingdom, December 13—17, 2014, P. 63—74.

Using Program Patterns for Parallel Programming. Base Concepts

V. A. Galatenko, galat@niisi.ras.ru, **N. I. Viukova,** niva@niisi.ras.ru, **K. A. Kostyukhin,** kost@niisi.ras.ru, Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences", Moscow, 117218, Russian Federation

Corresponding author:

Kostyukhin Konstantin A., Senior Researcher, Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences", Moscow, 117218, Russian Federation,
E-mail: kost@niisi.ras.ru

Received on January 09, 2019
Accepted on January 29, 2019

Program patterns as a tool of programming knowledge accumulation and usage are considered in this article in the context of parallel programming. They can serve as a basis for tests for evaulating performance of architectures and/or productivity of programmers who develop new parallel applications or parallelize legacy code. Program patterns can be useful for developing and implementing hardware that supports concurrency. Program patterns are an effective tool for learning parallel programming. Performance measurement tests can be developed based on program patterns. The development, implementation and maintenance of parallel programs designed for distributed, heterogeneous configurations is a complex problem. The usage of program patterns helps to maintain programmers' productivity at an economically acceptable level. This paper is the first part of work that contains results of analysis on program patterns implementations in various heterogeneous hardware-software systems. The authors will give the main concepts and definitions of program patterns and review existing architecture solutions such as program patterns for parallelizing on the operating systems level, distributed heterogeneous systems level and multikernel systems level. The second part of the work will consider the main mechanisms of applying program schemes when working with parallel systems.

Keywords: *program patterns, parallel programming, distributed systems, review, heterogeneous systems, multikernel systems, parallelizing methods*

For citation:

Galatenko V. A., Viukova N. I., Kostyukhin K. A. Using Program Patterns for Parallel Programming. Base Concepts, *Programmnaya Ingeneria*, 2019, vol. 10, no. 4, pp.147—159.

DOI: 10.17587/prin.10.147-159

References

1. **Betelin V. B., Galatenko V. A.** ESCORT — instrumentalnaya sreda programmirovaniya (Instrumental programming environment ESCORT), *Iubileinyi sbornik trudov institutov Otdeleniya informatiki RAN*, 1993, vol. 2, pp. 77—97 (in Russian).
2. **Galatenko V. A.** *Programmirovanie v standarte POSIX* (POSIX standard programming) / Eds. by V. B. Betelin, Moscow, INTUIT.RU, 2004, 560 p (in Russian).
3. **Galatenko V. A.** *Programmirovanie v standarte POSIX. Chast 2.* (POSIX standard programming. Part 2) / Eds by V. B. Betelin, Moscow, INTUIT.RU, 2004, 424 p. (in Russian).
4. **Mattson T., Sanders B., Massingill B.** *Patterns for parallel programming*, Addison-Wesley Professional, 2004, 384 p.
5. **Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, Zhen Xiao.** Spartan: A Distributed Array Framework with Smart Tiling, *Proc. of the 2015 USENIX Annual Technical Conference (USENIC ATC 15)*, Santa Clara, CA, USA, 2015, pp. 1—15.
6. **Brown K. J., Lee H. J., Rompf T., Sujeeth A. K., De Sa C., Aberger C., Olukotun K.** Have Abstraction and Eat Performance. Optimized Heterogeneous Computing with Parallel Patterns, *Proc. of CGO'16,* Barcelona, Spain, March 12—18, 2016, pp. 194—205.
7. **Danelutto M., De Matteis T., Mencagli G., Torquati M.** A Divide-and-Conquer Parallel Pattern Implementation for Multicores, *Proc. of SEPS'16,* Amsterdam, Netherlands, November 1, 2016, pp. 10—19.
8. **Khammassi N., Le Lann J.-C.** A High-Level Programming Model to Ease Pipeline Parallelism Expression On Shared Memory Multicore Architectures, *Proc. of HPC'14*, Tampa, Florida, April 13—16, 2014, article 9.
9. **Intel** Corporation, Threading Building Blocks, TBB, available at: http://www.threadingbuildingblocks.org (accessed 23.01.2019).
10. **Lee H. J., Brown K. J., Sujeeth A. K., Rompf T., Olukotun K.** Locality-Aware Mapping of Nested Parallel Patterns on GPUs, *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, United Kingdom, December 13—17, 2014, pp. 63—74.

УДК 004.2 DOI: 10.17587/prin.10.160-166

В. В. Корнеев, д-р техн. наук, проф., гл. науч. сотр., **И. Е. Тарасов,** д-р техн. наук, консультант, ФГУП "Научно-исследовательский институт "Квант", Москва, info@rdi-kvant.ru

Особенности архитектуры массово-параллельных проблемно-ориентированных СБИС

Проведен сравнительный анализ архитектур вычислительных устройств, в том числе процессорного типа, для организации массово-параллельных вычислений с оценкой производительности, эффективности использования площади кристалла СБИС и потребляемой мощности. Представлен подход к повышению продуктивности разработки проблемно-ориентированных СБИС, основанный на совместной оптимизации программных и аппаратных компонентов системы с учетом требований, которые формируются уменьшением проектных норм до 28 нм.

Ключевые слова: *СБИС, архитектура, процессор, вычисление*

Введение

Развитие САПР и контрактных производств СБИС предоставило специалистам разных прикладных областей возможность создавать проблемно-ориентированные сверхбольшие интегральные схемы (П-СБИС) для построения вычислительных систем, имеющих высокую эффективность на отдельных классах задач. Архитектура П-СБИС разрабатывается с ориентацией на реализацию алгоритмов в отдельных областях их специализации. Однако современные технологии микроэлектроники вносят свои ограничения в создаваемые архитектуры.

В настоящее время исследователями рассматривается широкий круг вопросов, связанных с проектированием СБИС, как на уровне общей архитектуры [1—4], так и для специализированных решений [5—8]. При этом в ряде работ учитываются характеристики элементной базы и топологических библиотек, а также их влияние на архитектуру проектируемой СБИС [9—11]. Все более важную роль играют моделирование на системном уровне и совместная оптимизация аппаратной архитектуры и программного обеспечения [12, 13].

В работе предпринята попытка рассмотрения последствий учета архитектурных, программных и топологических ограничений при выборе архитектуры П-СБИС на всех уровнях проектирования, включая системный уровень, уровни архитектуры вычислительных устройств и архитектуры вычислительного элемента.

Выбор архитектуры П-СБИС

Проектирование массово-параллельных П-СБИС большого логического объема подразумевает создание иерархии входящих в ее состав гетерогенных компонентов. Выбор неудачного способа соединения компонентов и алгоритмов их взаимодействия может существенно ухудшить общие характеристики П-СБИС, сформировав "узкие места". Поэтому важной задачей проектирования П-СБИС является выбор ее архитектуры, что требует согласованного представления этой архитектуры на всех уровнях проектирования.

Выбор архитектуры П-СБИС на системном уровне

На системном уровне определяются порядок взаимодействия П-СБИС и внешних по отношению к ней устройств, соотношение числа операций на кристалле П-СБИС и объема передаваемых по внешним интерфейсам данных. На основании этого соотношения определяются требования к характеристикам вычислительных и коммуникационных устройств. Рост степени интеграции обусловливает возможность реализации в одной СБИС функций многих СБИС меньшей степени интеграции, а также реализацию как интерфейсов между ними, так и их внешних интерфейсов. Однако состав компонентов и интерфейсов в такой СБИС, как правило, должен быть иным для рациональной реализации требуемых функций в силу следующих особенностей:

— опережающего роста производительности вычислений по сравнению с ростом производительности коммуникаций, затрудняющего реализацию ускорителей вычислений, ориентированных на непрерывное получение исходных данных и отправку результатов;

— усиливающегося по мере роста степени интеграции энергопотребления интегральных схем, существенно определяемого суммарной длиной проводников, доставляющих команды и данные к функциональным устройствам;

— ограничения площади локальных кластеров на СБИС, состоящих из функциональных устройств и источников команд и операндов для них, использу-

ющих локально синхронно-тактовое дерево, а также переход к архитектурам класса GALS (*Globally Asynchronous*, *Locally Synchronous*), в которых глобальная асинхронность совокупности кластеров на кристалле предусматривает ресинхронизацию данных при их передаче между синхронными вычислительными устройствами разных кластеров.

Существующая тенденция увеличения плотности компонентов на кристалле приводит на практике к тому, что производительности внешних интерфейсов недостаточно для обеспечения потока данных к параллельно работающим высокопроизводительным вычислительным узлам, если обработка данных этими узлами происходит за небольшое число тактов. Например, для программируемых логических интегральных схем (ПЛИС) Xilinx Versal, анонсированных к выпуску с технологическими нормами 7 нм, суммарная производительность оценивается в 49...147 Топ/с (10^{12} операций в секунду) над целочисленными 8-битными данными [14], тогда как пиковая производительность коммуникационной подсистемы составляет 2,9 Тбит/с. Этот факт означает, что при передаче одного байта СБИС Versal за это же время выполняет 135...400 операций. Таким образом, реализация простого ускорителя, работающего по принципу "прием данных — операция — отправка данных", в настоящее время сталкивается с существенными ограничениями, которые усугубляются по мере увеличения плотности компонентов на кристалле. Поэтому при выборе архитектуры П-СБИС предлагается ориентироваться на гетерогенные вычислительные архитектуры. При реализации таких архитектур компоненты, генерирующие операнды для вычислительных устройств, являются частью П-СБИС и интегрированы в ту же систему-на-кристалле, что и вычислительные устройства.

Выбор архитектуры вычислительного модуля

На уровне вычислительного устройства выбираются архитектура вычислительного модуля, выполняемого в рамках отдельного тактового дерева, архитектура его узлов, а также способ взаимодействия модуля с асинхронными по отношению к нему компонентами П-СБИС. На этом уровне необходимо учесть:

— большую параллельность обработки данных, обусловленную использованием большого числа вычислительных модулей, состоящих из функциональных блоков, регистров и локальных по отношению к использующим их функциональным блокам блоков памяти;

— программируемость коммутационной структуры, объединяющей совокупность одинаковых вычислительных модулей или образующей иерархию коммуникационных сетей, на верхнем уровне которой управляющий вычислительный модуль объединяется с совокупностью кластеров модулей следующего уровня иерархии, а также с каналами ввода-вывода и внешней памятью;

— иерархическую структуру памяти, нижний уровень которой образуют небольшие блоки памяти, обслуживающие функциональные блоки, следующий уровень состоит из блоков памяти, служащих для хранения данных и команд, пересылаемых между блоками памяти нижнего по отношению к рассматриваемому уровня и блоком памяти верхнего уровня;

— потенциальную избыточность задействованных ресурсов при реализации универсальных вычислительных модулей;

— неравнозначность некоторых задач топологического проектирования, например, сложность реализации комплексных узлов, таких как умножители, полные коммутаторы и т. п., планирование размещения компонентов памяти и учет задержки распространения сигнала внутри этих блоков;

— уровень программируемости и возможного объединения ресурсов внутри модуля;

— проблему "темного кремния", заключающуюся в схемотехнической возможности формирования схемы, которая при ее реализации на кристалле СБИС приведет к превышению предельно допустимой рассеиваемой мощности для полупроводниковой пластины [15].

С учетом того что, по проведенным оценкам, технологический процесс 28HPC+ фабрики TSMC (Taiwan Semiconductor Manufacturing Company) может генерировать до 5 Вт/мм^2 при синтезе цифровых вычислительных узлов, а технически допустимым с точки зрения отвода теплоты от кристалла СБИС является диапазон 0,5...0,7 Вт/мм^2, необходимо предусмотреть целый комплекс мер по обеспечению снижения потребляемой мощности как на технологическом, так и на архитектурном уровнях. При этом важным является тот факт, что отключение питания или тактового сигнала части устройств хотя и приводит к снижению потребляемой мощности, является неэффективным из практических соображений, так как означает остановку части устройств СБИС и пропорциональное снижение общей производительности. Более перспективным представляется выбор архитектур, у которых решение задачи изначально подразумевает работу только части размещенных на кристалле схемотехнических узлов. Соотношение между предельно допустимым и технически реализуемым при плотной компоновке схемы потреблением мощности позволяет оценить приблизительное соотношение между активной и общей частями устройств на кристалле. Приведенные для технологического процесса 28HPC+ показатели дают приблизительную оценку этого соотношения — 0,05...0,1. Такая оценка означает, например, что на архитектурном уровне вычислительные модули П-СБИС могут состоять из 10...20 и более регистров, объединенных в блок (в котором активным является только один из регистров), и 10...20 вычислительных операций в арифметико-логическом устройстве (из которых также активной является только одна).

Далее рассмотрим ряд представленных архитектур вычислительных модулей, предлагаемых для ана-

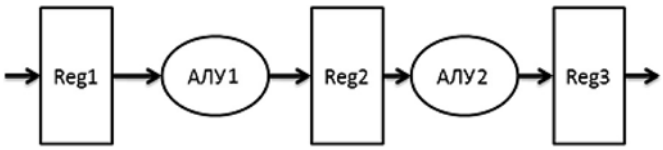


Рис. 1. Конвейерный вычислитель:
Reg1, Reg2, Reg3 — регистры между ступенями конвейера; АЛУ1, АЛУ2 — арифметико-логические устройства

лиза в качестве возможных решений для проблемно-ориентированных СБИС с массовым параллелизмом вычислений.

Наиболее простой архитектурой является конвейерный вычислитель, структура которого показана на рис. 1. Конвейерный вычислитель позволяет реализовать сложную последовательную обработку данных и хорошо соответствует современному подходу к синхронному проектированию, позволяющему получать высокие тактовые частоты и предсказуемо надежную работу такой схемы при ее топологической реализации.

С точки зрения схемотехники конвейерный вычислитель является эффективным решением, поскольку обладает невысокой функциональной избыточностью. При реализации П-СБИС, содержащих сотни конвейерных вычислителей, окажется достижимой производительность в сотни Топ/с, близкая к производительности специализированных СБИС (С-СБИС) с прямой аппаратной реализацией алгоритмов проблемной области. Однако это решение приводит к тому, что такой вычислитель имеет высокую плотность активных компонентов и генерирует чрезмерную мощность на единицу площади СБИС. Возможным подходом к разрешению этого вопроса является намеренное снижение плотности активных компонентов или размещение в П-СБИС конвейерных вычислителей в комбинации с узлами, которые заведомо неактивны в процессе работы такого вычислителя. Однако это приводит к снижению общей производительности на единицу площади, так как часть кристалла вынужденно простаивает. Дополнительно при топологическом проектировании необходимо принимать меры для недопущения локального нагрева фрагментов СБИС с сосредоточением активных компонентов. По этим причинам гипотетическая, физически не реализуемая С-СБИС, должна быть заменена на настраиваемую (конфигурируемую) или программируемую П-СБИС, в связи с чем возникает вопрос, что считать лучшей реализацией специализированной схемы?

Конфигурируемый конвейерный вычислитель, показанный на рис. 2, частично разрешает проблемные вопросы конвейерного вычислителя за счет размещения между регистровыми устройствами коммутируемых (конфигурируемых) арифметико-логических устройств. Такое решение, с одной стороны, обусловливает некоторую функциональную избыточность, с другой стороны, гарантирует, что в блоке арифметико-логических устройств только одно из них будет активным, а следовательно, и будет генерировать повышенное тепловыделение.

Общим ограничением конвейерных архитектур является их ориентация на алгоритмы определенного типа, в которых число стадий обработки данных и их виды могут быть заранее определены на этапе проектирования П-СБИС. Невыполнение этого условия может приводить как к снижению эффективности использования площади СБИС, так и к отсутствию возможности реализации алгоритмов, не укладывающихся в реализованную структуру вычислителя.

В конфигурируемых устройствах пути доставки операндов и результатов содержат коммутаторы, реализуемые обычно как мультиплексоры. При этом возникает DPM-проблемный вопрос (*DataPath Merging*) [16], требующий разрешения. Коммутаторы используются для переключения потока данных между электронными блоками, образующими устройство обработки. Введение коммутаторов, с одной стороны, делает возможным экономную реализацию большого числа алгоритмов за счет использования общих для этих алгоритмов блоков. С другой стороны, сами коммутаторы и пути от блоков к/из коммутаторов служат источником дополнительных задержек и потребления энергии. Таким образом, при построении конфигурируемых вычислителей важно, чтобы выигрыш от экономии оборудования превышал потери, вызванные снижением тактовой частоты вследствие задержек и энергетических ограничений.

Переход к программируемым вычислительным устройствам позволяет существенно расширить круг решаемых с использованием П-СБИС задач и устранить ограничение на заранее определенные структуры алгоритмов. Структурная схема RISC-процессора, т. е. процессора с сокращенным набором команд — *Reduced Instruction Set Computer*, показана на рис. 3. В ней наиболее важным признаком является последовательное выполнение команд с модификацией одного и того же регистрового блока, обозначенного на рис. 3 как Regs. Для RISC-процессора, который может быть представлен как программируемый конечный автомат, ограничениями могут служить не-

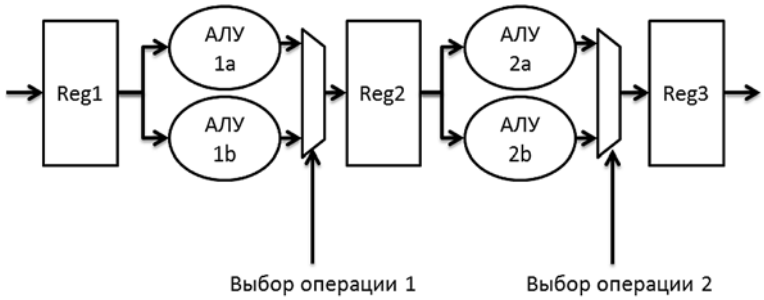


Рис. 2. Конфигурируемый конвейерный вычислитель:
АЛУ 1a, АЛУ 1b — выбираемое АЛУ первой ступени конвейера; АЛУ 2a, АЛУ 2b — выбираемое АЛУ второй ступени конвейера

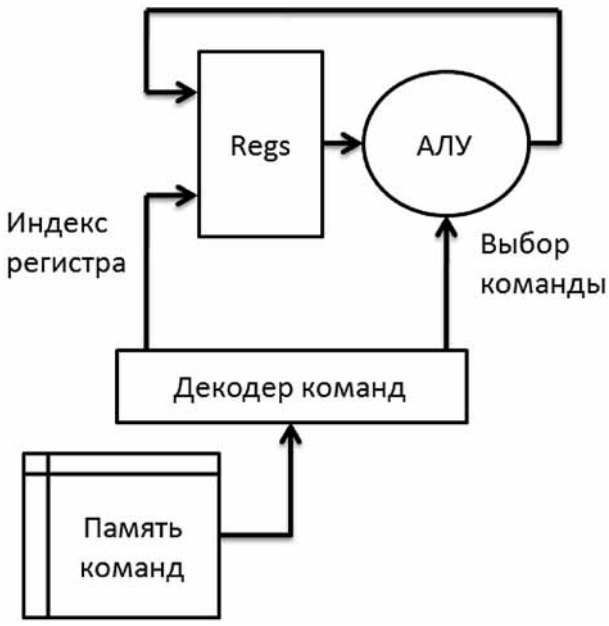


Рис. 3. RISC-процессор:
Regs — регистровый блок

обходимый для реализации алгоритма объем памяти и функциональный состав поддерживаемых команд. Тем не менее эти параметры могут быть выбраны для достаточно широкого подкласса задач. Однако для RISC-процессора необходимо добавление к вычислительным устройствам также устройства управления, памяти команд и избыточного набора регистров, гарантированно достаточного для реализации любого из алгоритмов заданного класса. Такой подход обусловливает увеличение площади RISC-процессора по сравнению с непрограммируемым устройством аналогичного назначения. Вместе с тем избыточность ресурсов положительно влияет на удельное энергопотребление СБИС, так как в каждый момент времени активной, т. е. потребляющей энергию, является только часть компонентов RISC-процессора.

Сокращения доли дополнительных расходов на организацию вычислений можно добиться, используя архитектуру SIMD (*Single Instruction, Multiple Data*) для массива RISC-процессоров. Это позволит обобщить память команд и частично устройство управления.

Увеличение количества выполняемых RISC-процессором операций можно обеспечить, переходя к многопортовой организации регистрового файла. Вместе с этим следует увеличить и число программируемых вычислительных узлов, реализуя таким образом явное управление параллельными вычислениями (EPIC — *Explicitly Parallel Instruction Computing*). Архитектура EPIC подразумевает длинное командное слово, управляющее несколькими операциями одновременно, и является разновидностью более общего подхода VLIW (*Very Long Instruction Word*), внедренной компаниями Hewlett-Packard и Intel. Структурная схема процессорного элемента с явным параллелизмом операций показана на рис. 4. Ожидаемым преимуществом такого подхода является частичное обобщение ресурсов, требуемых для выполнения нескольких операций. Однако для EPIC необходимо на уровне компилятора решить задачу распределения нагрузки между параллельными вычислительными узлами, чтобы избежать их простоя. Ряд алгоритмов, кроме того, могут не допускать распараллеливания. Таким образом, для архитектуры EPIC важное значение приобретает моделирование выполняемых алгоритмов на системном уровне и оптимизация структуры вычислительных узлов, регистровых файлов и связей между ними.

Ограничивающими факторами для процессора с несколькими взаимодействующими потоками данных в пределах одного ядра является, во-первых, объективное ограничение на операции, которые могут быть выполнены параллельно в целевых алгоритмах. Отсутствие операндов для команды, которая могла бы быть выполнена на втором и последующем трактах данных, может быть вызвано как особенностями алгоритма, так и наличием конфликта по доступу к ресурсам ядра. Во-вторых, наличие взаимосвязей между N регистровыми и M функциональными узлами (АЛУ), где N и M в общем случае могут быть не равны, требует добавления коммутаторов, усложняющих топологическую реализацию такого узла.

В программируемых вычислительных устройствах важно снизить до возможного минимума энергетические потери и задержки на выборку и дешифрацию команд, подготовку операндов. Это может быть достигнуто размещением небольшой регистровой памяти команд непосредственно вблизи обрабатывающих арифметико-логических устройств [17,

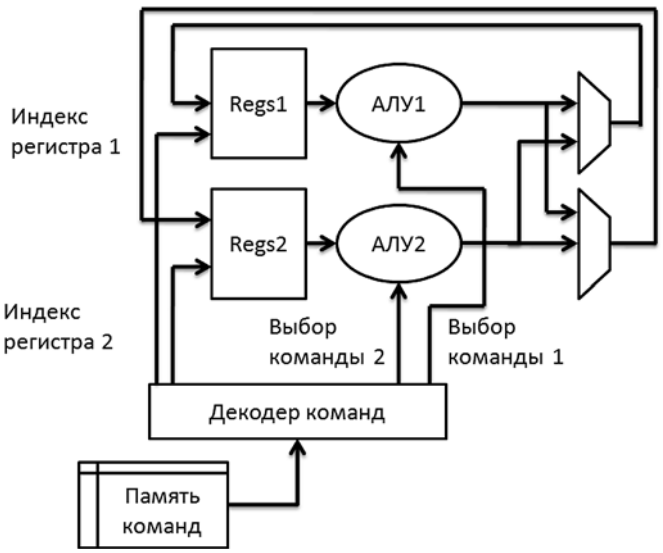


Рис. 4. Процессор с явным параллелизмом операций:
Regs1, Regs2 — регистровые блоки

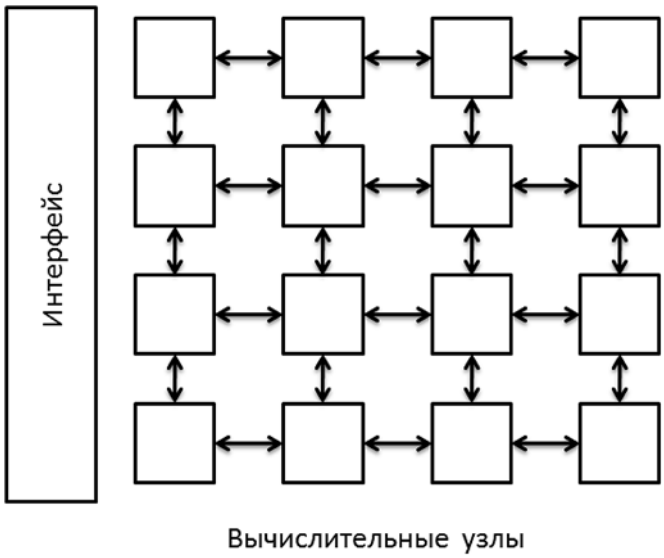


Рис. 5. Сеть вычислительных узлов

18], использующих операнды из блоков локальной, близко расположенной памяти данных. Последнее приводит к организации многоуровневой памяти и иерархической структуры П-СБИС в целом.

Сеть вычислительных узлов (рис. 5) сочетает пространственное и временное распределение выполнения операций одной задачи. Для такой сети, узлы которой могут быть как RISC-, так и EPIC-процессорами, появляется возможность реализации как конвейеризованных вычислений, когда данные передаются от одного узла к другому, так и последовательных вычислений, когда операции выполняются на одном узле.

Для сети вычислительных узлов возможно возникновение ситуации, когда часть узлов простаивает ввиду отсутствия данных для обработки, что, в свою очередь, может быть вызвано неэффективностью реализованной коммутационной сети.

Таким образом, вычислительные архитектуры могут быть описаны числом вычислительных узлов N и числом тактов M, которые необходимы для выполнения задачи. Для RISC-процессора $N = 1$, а для полностью конвейеризованной схемы $M = 1$. Варианты промежуточных решений соответствуют рассмотренным в статье архитектурам VLIW/EPIC, конфигурируемым трактам данных и сети вычислительных узлов. Однако такие решения не ограничиваются ими.

Дополнительным архитектурным параметром является показатель параллельности по данным, который может быть эффективным при реализации процессорных устройств. Применение подхода SIMD к приведенным на рис. 3—5 процессорным архитектурам позволяет использовать одну память программ и одно устройство управления для нескольких потоков данных. Такой подход сокращает избыточность ресурсов вычислительного модуля при реализации потока управления.

Проектирование вычислительного узла

На уровне вычислительного элемента необходимо учесть:

— потенциальную избыточность задействованных ресурсов при реализации узлов вычислительного элемента;

— вопросы совместной оптимизации аппаратного и программного обеспечения.

В работе [19] рассмотрено унифицированное описание вычислительного узла четверкой параметров (I, O, D, S), где

I — число инструкций, выполняемых за один такт;

O — число операций, определяемое инструкцией;

D — число операндов (пар операндов), относящихся к операциям;

S — степень конвейеризации.

Для SIMD-архитектур $D > 1$, а для MIMD (т. е. *Multiple Instruction, Multiple Data* — "много инструкций, много данных") $I > 1$, $D > 1$. Для массового параллелизма можно дополнительно рассматривать параллелизм трактов обработки данных (*datapath*) на уровне отдельного вычислительного узла, что позволяет применить архитектурное описание узла в виде четверки (R, $<O>$, $<D>$, $<S>$), где

R — число независимо вычисляемых результатов;

$<O>$ — вектор числа операций, выполняемых инструкцией для каждого из трактов данных;

$<D>$ — вектор числа операндов (пар операндов) для каждого из трактов данных;

$<S>$ — вектор конвейеризации для каждого из трактов данных.

В процессе проектирования вычислительного узла необходимо разрешить противоречие между двумя отмеченными далее тенденциями.

1. В целях повышения производительности узла необходимо увеличивать число трактов данных, операций и операндов.

2. Увеличение этих параметров ведет к увеличению площади, энергопотребления и задержки распространения сигнала.

Достижение оптимальных характеристик узла проводится итеративным методом путем перебора архитектурных моделей-кандидатов, построения для них псевдокода и последующей оценки характеристик аппаратной реализации в выбранном технологическом базисе.

С учетом необходимости выполнения операций общего назначения (сложение, вычитание, поразрядные логические операции) структура вычислительного узла может выглядеть, как показано на рис. 6.

Все представленные на рис. 6 компоненты вычислительного узла имеют мультиплексируемый выход, управляемый соответствующим полем кода команды (или конфигурирующего машинного слова). Такое решение позволяет на каждом такте выбрать произвольную операцию для каждого компонента. Таким образом, показанный вычислительный узел с учетом дополнительных компонентов реализует несколько арифметических или логических операций за один такт.

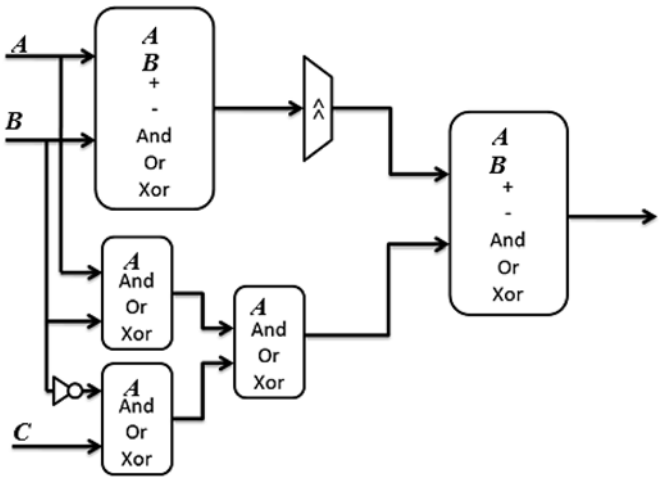


Рис. 6. Пример вычислительного узла

Можно обратить внимание на поддерево из трех вычислительных компонентов, реализующих только поразрядные логические операции (для входов *B* и *C*). Особенность этого поддерева заключается в том, что задержка распространения сигнала для поразрядных операций обусловлена задержкой на одном логическом вентиле (около 10...30 пс), тогда как задержка на сумматоре в верхнем компоненте (для входов *A* и *B*) оказывается больше в силу необходимости распространения сигнала переноса (около 150—200 пс). Поэтому реализация дополнительного поддерева не образует критического пути с точки зрения времени распространения сигнала. Таким образом, данный подход использует особенности топологических библиотек для техпроцесса 28HPC и подобных ему, что позволяет реализовывать в вычислительных узлах разветвленные структуры, выполняющие более одной операции за такт.

Предлагаемый в настоящей работе порядок проектирования вычислительного узла заключается в следующем:

1) выбор модели архитектуры в пространстве (*R*, <*O*>, <*D*>, <*S*>);

2) компиляция псевдокода по набору модельных задач, выявление наиболее часто встречаемых последовательностей операций и создание вычислительных узлов для их однотактного исполнения, проверка результата с помощью эмулятора и создание узлов процессорного элемента на уровне регистровых передач (т. е. RTL-описания);

3) проверка синтезируемости полученной схемы и выполнение функциональной верификации на системном уровне;

4) выполнение оценки топологической реализации в выбранном технологическом базисе.

Оптимизация архитектуры П-СБИС с учетом характеристик инструментального программного обеспечения обусловливает применение подхода, при котором анализ эффективности компилятора и анализ характеристик аппаратной платформы производятся различными инструментами и требуют отдельной стадии сопоставления.

Заключение

Применение системного моделирования с предварительной оценкой достижимых характеристик аппаратной платформы (площадь, тактовая частота, энергопотребление) позволяет получить набор вариантов архитектуры, включая число узлов, объем памяти, состав команд, структуру АЛУ. Такой подход является основой для разработки соответствующих модулей или параметризации готовых решений.

Список литературы

1. **Таненбаум Э., Остин Т.** Архитектура компьютера. 6-е изд. СПб.: Питер, 2013. 816 с.

2. **Харрис Д. М., Харрис С. Л.** Цифровая схемотехника и архитектура компьютера. Второе издание. Morgan Kaufman © English Edition, 2013. 1676 p.

3. **Hennessy J. L., Patterson D. A.** Computer Architecture. 6th Edition. A Quantitative Approach. The Morgan Kaufmann Series in Computer Architecture and Design, 2017. 936 p.

4. **Abd-El-Barr M., El-Rewini H.** Fundamentals of Computer Organization and Architecture, A JOHN WILEY & SONS, INC PUBLICATION, 2005. 290 p.

5. **Falk H.** Decisive Aspects in the Evolution of Microprocessors// Proceedings of the IEEE. 2004. Vol. 92. P. 1895.

6. **Abadi M., Barham P., Chen J.** et al. TensorFlow: A system for large-scale machine learning // Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16). November 2—4, 2016, Savannah, GA. 2016. P. 265—283.

7. **Nurvitadhi E., Sheffield D., Sim J.** et al. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC // International Conference on Field-Programmable Technology (FPT), 2016. P. 77—84.

8. **Magaki I., Khazraee M., Gutierrez L. V., Taylor M. B.** ASIC Clouds: Specializing the Datacenter // ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016. P. 178—190.

9. **Krpic Z., Horvat G., Zagar D., Martinovic G.** Towards an energy efficient SoC computing cluster // 37th International Convention on Information and Communication Technology, Electronics and Microelectronics, (MIPRO), 2014. P. 178—182.

10. **Shien-Yang Wu, Lin C. Y., Chiang M. C. et al.** A 16nm FinFET CMOS technology for mobile SoC and computing applications // IEEE International Electron Devices Meeting, 2013. P. 9.1.1—9.1.4.

11. **Swamynathan Sankaranarayanan, Banumathi V.** Design and analysis of FPGA based 32 bit ALU using reversible gates // IEEE International Conference on Electrical, Instrumentation and Communication Engineering (ICEICE), 2017. P. 1—4.

12. **Villarraga C., Stoffel D., Kunz W.** Software in a hardware view: New models for HW-dependent software in SoC verification and test // 2014 International Test Conference. 2014. P. 1—9.

13. **Ganesh R.** Design Issues in Hardware / Software Co-Design // International Journal of Research in Electronics & Communication Technology. 2014. Vol. 2, Issue1. P. 1—5.

14. **Versal** Architecture and Product Data Sheet: Overview DS950 (v1.0) October 2, 2018 Advance Product Specification. URL: https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf

15. **Esmaeilzadeh H., Blem E., Amantet R. St.** et al. Dark Silicon and the End of Multicore Scaling // IEEE Micro. 2011. Vol. 32. P. 122—134.

16. **York J., Chiou D.** On the Asymptotic Costs of Multiplexer-based Recongurability // DAC '12. Proceedings of the 49th Annual Design Automation Conference, Jun 03—07, 2012. San Francisco, CA, USA, 2012. P. 790—795.

17. **Balfour J., Dally W. J., Black-Schaffer D.** et al. An Energy-Efficient Processor Architecture for Embedded Systems // IEEE Computer architecture letters. 2008. Vol. 7, No. 1. P. 29—32.

18. **Japan's** own processor "PEZY-SC2" and equipped with supercomputers "Gyoko" Details. URL: https://pc.watch.impress.co.jp/docs/news/1091458.html

19. **Jouppi N. P., Wall D. W.** Available Instruction-Level Parallelism for Superscalar and Super-pipelined Machines // Proceedings of the 3rd International Conference on Architectural Support, 1989. P. 272—282.

Peculiar Properties of Architecture of Parallel Application Specific Integral Circuit

V. V. Korneev, korv@rdi-kvant.ru, **I. E. Tarasov**, ilya_e_tarasov@mail.ru,
Research and Development Institute 'Kvant', Moscow, 125438, Russian Federation,

Corresponding author:

Korneev Victor V., Principal Researcher, Research and Development Institute "Kvant", Moscow, 125438, Russian Federation.
E-mail: korv@rdi-kvant.ru

Received on December 03, 2018
Accepted on February 01, 2019

A comparative analysis of computing device architectures, including those of a processor type, is given for organizing mass-parallel computing with an assessment of performance, efficient use of VLSI chip area and power consumption. Several architectures of computing devices intended for the implementation of parallel computing, including both non-programmable and programmable devices, are considered. The analysis shows that the considered architectures have a number of features that limit their use in problem-oriented VLSI, such as limited functionality, increased power consumption, inefficient use of the chip area, inefficient use of computing nodes in the process of algorithms. The process of designing problem-oriented VLSI should be performed using complex optimization according to the parameters determined at various design stages — from the choice of architecture to the topological implementation of VLSI. An approach to increase the productivity of development of problem-oriented VLSI based on the co-optimization of software and hardware components of the system with the requirements that are formed by reducing the design standards to 28 nm is presented. The approach is based on the use of system modeling, using both VLSI emulation at the level of the program model, and modeling at the register transfer level, with the parameters used to assess the quality of the project with respect to the selected optimality criteria. The use of system modeling with a preliminary assessment of the achievable characteristics of the hardware platform (area, clock frequency, power consumption) allows one to get a set of architecture options, including the number of nodes, memory size, instruction set, ALU structure. This approach is the basis for the development of appropriate modules or parameterization of ready-made solutions.

Keywords: *VLSI, architecture, processor, computation*

For citation:

Korneev V. V., Tarasov I. E. Peculiar Properties of Architecture of Parallel Application Specific Integral Circuit, *Programmnaya Ingeneria,* 2019, vol. 10, no. 4, pp. 160—166.

DOI: 10.17587/prin.10.160-166

References

1. **Tanenbaum A. S., Austin T.** *Structured Computer Organization*, 6th Edition, Amsterdam, The Netherlands, Vrije University, 2013, 800 p.
2. **Harris D., Harris S.** *Digital Design and Computer Architecture,* 2nd Edition, Morgan Kaufman, English Edition, 2013, 1676 p.
3. **Hennessy J. L., Patterson D. A.** *Computer Architecture*, 6th Edition, A Quantitative Approach, The Morgan Kaufmann Series in Computer Architecture and Design, 2017, 936 p.
4. **Abd-El-Barr M., El-Rewini H.** *Fundamentals of Computer Organization and Architecture*, A JOHN WILEY & SONS, INC PUBLICATION, 2005, 290 p.
5. **Falk H.** Decisive Aspects in the Evolution of Microprocessors, *Proceedings of the IEEE,* 2004, vol. 92, pp. 1895.
6. **Abadi M., Barham P., Chen J.,** et al. TensorFlow: A system for large-scale machine learning, *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16),* November 2—4, 2016, Savannah, GA, 2016, pp. 265—283.
7. **Nurvitadhi E., Sheffield D., Sim J.** et al. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC, *International Conference on Field-Programmable Technology (FPT),* 2016, pp. 77—84.
8. **Magaki I., Khazraee M., Gutierrez L. V., Taylor M. B.** ASIC Clouds: Specializing the Datacenter, *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA),* 2016, pp. 178—190.
9. **Krpic Z., Horvat G., Zagar D., Martinovic G.** Towards an energy efficient SoC computing cluster, *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, (MIPRO),* 2014, pp. 178—182.
10. **Shien-Yang Wu, Lin C. Y., Chiang M. C.** et al. A 16nm FinFET CMOS technology for mobile SoC and computing applications, *IEEE International Electron Devices Meeting,* 2013, pp. 9.1.1—9.1.4.
11. **Swamynathan Sankaranarayanan, Banumathi V.** Design and analysis of FPGA based 32 bit ALU using reversible gates, *IEEE International Conference on Electrical, Instrumentation and Communication Engineering (ICEICE),* 2017, pp. 1—4.
12. **Villarraga C., Stoffel D., Kunz W.** Software in a hardware view: New models for HW-dependent software in SoC verification and test, *2014 International Test Conference,* 2014, pp. 1—9.
13. **Ganesh R.** Design Issues in Hardware / Software Co-Design, *International Journal of Research in Electronics & Communication Technology*, 2014, vol. 2, issue 1, pp. 1—5.
14. **Versal** Architecture and Product Data Sheet: Overview DS950 (v1.0) October 2, 2018 Advance Product Specification, available at: https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf
15. **Esmaeilzadeh H., Blem E., Amantet R. St.** et al. Dark Silicon and the End of Multicore Scaling, *IEEE Micro*, 2011, vol. 32, pp. 122—134.
16. **York J., Chiou D.** On the Asymptotic Costs of Multiplexer-based Recongurability, *DAC '12. Proceedings of the 49th Annual Design Automation Conference*, Jun 03—07 2012 San Francisco, CA, USA, 2012, pp. 790—795.
17. **Balfour J., Dally W. J., Black-Schaffer D.,** et al. An Energy-Efficient Processor Architecture for Embedded Systems, *IEEE Computer architecture letters*, 2008, vol. 7, no. 1, pp. 29—32.
18. **Japan's** own processor "PEZY-SC2" and equipped with supercomputers "Gyoko" Details, available at: https://pc.watch.impress.co.jp/docs/news/1091458.html
19. **Jouppi N. P., Wall D. W.** Available Instruction-Level Parallelism for Superscalar and Super-pipelined Machines, *Proceedings of the 3rd International Conference on Architectural Support,* 1989, pp. 272—282.

УДК 004.4 DOI: 10.17587/prin.10.167-177

В. Л. Макаров[1], академик РАН, науч. руководитель, e-mail: director@cemi.rssi.ru, makarov@cemi.rssi.ru, **А. Р. Бахтизин**[1], член-корр. РАН, д-р экон. наук, проф., директор, e-mail: albert.bakhtizin@gmail.com, **Г. Л. Бекларян**[1], канд. экон. наук, ст. науч. сотр., e-mail: glbeklaryan@gmail.com, **А. С. Акопов**[1,2], д-р техн. наук, проф., гл. науч. сотр., e-mail: akopovas@umail.ru,
[1] Центральный экономико-математический институт РАН (ЦЭМИ РАН), Москва
[2] Национальный исследовательский университет "Высшая Школа Экономики", Москва

Разработка программной платформы для крупномасштабного агент-ориентированного моделирования сложных социальных систем*

Представлен подход к проектированию программной платформы, предназначенной для крупномасштабного агент-ориентированного имитационного моделирования сложных социальных систем. Подобные системы требуют применения как суперкомпьютерных технологий и параллельных вычислений, так и феноменологических описаний поведения агентов и их взаимодействий. Разработана агентная модель поведения толпы в аэропорту, которая учитывает влияние различных факторов и позволяет определить наилучшие значения важнейших ресурсных характеристик аэропорта, обеспечивающих устранение (диссипацию) кластеров толпы.

Ключевые слова: *агентное моделирование, социальные системы, параллельные вычисления, модель толпы, суперкомпьютерные технологии, C++, MPI, программная платформа для имитационного моделирования*

Введение

В настоящее время в общественных науках возникает потребность в применении суперкомпьютерных технологий и параллельных вычислений, в частности, при проектировании и исследовании крупномасштабных агент-ориентированных моделей. В качестве примера сложной социально-ориентированной системы в настоящей работе рассматривается модель поведения толпы при отсутствии и наличии чрезвычайных ситуаций. Первые исследования, посвященные данной проблематике, были представлены в работе Минца [25], в ней описаны условия и причины возникновения паники, определяемые доминирующим влиянием коллективного бессознательного (люди как часть толпы действуют иначе, чем люди как индивиды).

К пионерским работам, направленным на исследование поведения толпы с использованием методов имитационного моделирования, следует отнести работы Дирка Хелбинга. В его исследовании 2000 г. в журнале Nature [21] впервые удалось воспроизвести ряд характерных для толпы явлений, таких как образование пробок, вовлечение новых людей в панику и другие, основанные только лишь на описании как поведения, так и взаимодействия агентов системы на молекулярном уровне. В предложенных Д. Хелбингом моделях [20—23] используются методы молекулярной динамики, в которых психологические и социальные факторы рассматриваются как потенциалы взаимодействия между молекулами-людьми. Такой подход, несмотря на многие преимущества, практически не применим для крупномасштабного агент-ориентированного моделирования вследствие высокой вычислительной сложности, связанной с ростом размерности подобной системы.

Такая сложность может быть преодолена в рамках феноменологического подхода (существенно более простое аналоговое описание состояний агентов и их взаимодействий). Этот подход был реализован в работах Л. Бекларяна и А. Акопова [2, 11]. В рамках подобного подхода априори определяются состояния

* Проект выполнен при частичной финансовой поддержке Российского фонда фундаментальных исследований (Грант РФФИ 18-29-03139-мк).

агентов с их характеристиками, правила взаимодействия агентов и правила принятия решений. В такой модели пространственная динамика агентов описывается с помощью системы дифференциальных уравнений с переменной структурой, учитывающей различные варианты взаимодействия агентов друг с другом и с простыми препятствиями. Одной из характеристик такой модели является определение понятия личного пространства агента, которое формирует явление "эффект турбулентности" толпы и панику. Преимуществом данного подхода является уменьшение вычислительной сложности определения координат перемещения агентов с сохранением точности поведения агентов. Однако и представленный выше подход характеризуется определенным упрощением системы принятия решений. Он не учитывает всю совокупность факторов, влияющих на выбор агентом наиболее предпочтительного пути и скорости перемещения. Поэтому в работе [5] феноменологический подход был уточнен за счет включения в модель индивидуальной системы принятия решений каждого агента, основанной на анализе множественных факторов. Например, присутствие других агентов в выбранном секторе, расстояние от стен, желание агента двигаться к выходу, нежелание агента отклоняться от первоначального направления движения и др. Такое уточнение является развитием системы принятия решений, представленной в работе Антонини [12]. Принятие решений в работе [5] базируется на анализе ситуации в рамках сектора обзора агента с углом развертки, составляющим примерно 170˚ (рис. 1). В результате агент в каждый момент времени осуществляет выбор той альтернативы, при которой достигается минимальное значение некоторого целевого функционала, учитывающего влияние множественных факторов.

Такое принятие индивидуальных решений требует точного анализа ситуации вокруг агента. Оно обусловливает необходимость иметь полную информацию о характеристиках всех остальных агентов рассматриваемой системы, например, данных об их текущих координатах, направлении и скорости движения и др. В результате, при использовании традиционных систем имитационного моделирования,

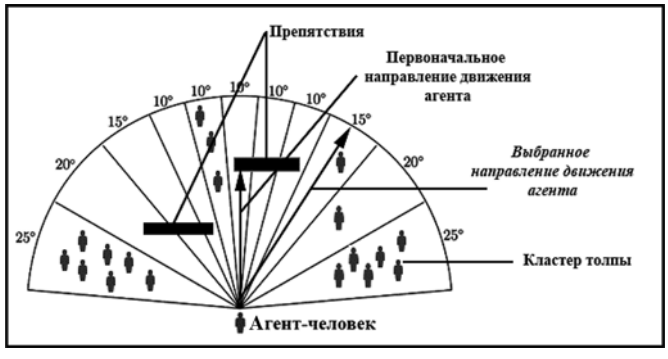


Рис. 1. Выбор агентом наиболее предпочтительной секции для перемещения

таких как AnyLogic, NetLogo и др. [1], с ростом числа агентов временная эффективность агент-ориентированных моделей существенно снижается вплоть до полной ее неработоспособности, а именно — отсутствия возможности запуска модели и получения результатов за полиномиальное время.

Одним из способов преодоления подобных трудностей является переход от динамики агентов к динамике агентских кластеров, предложенной в работе А. Бекларяна и А. Акопова [13]. Данный подход позволил существенно повысить временную эффективность моделей поведения толпы и исследовать сценарии оптимальной эвакуации людей при чрезвычайной ситуации с использованием интеллектуальных агентов-спасателей. При этом предложенный механизм формирования кластеров толпы, основанный на модифицированном алгоритме нечеткой кластеризации, является достаточно универсальным в том смысле, что он может учитывать всю совокупность факторов, влияющих на стремление агентов друг к другу. К числу таких факторов относятся, например, психология агентов, родительские связи между агентами, расстояния между ними, наличие общих целей и т. д.

Вместе с тем создание ряда сложных процессно-ориентированных моделей требует более детального описания динамики агентов, зависящей от сервисов и ресурсов. Так, например, при моделировании поведения толпы в аэропорту с учетом влияния различных факторов (геометрии внутреннего пространства, числа сотрудников, занятых обслуживанием пассажиров, и др.) снижение размерности модели за счет агентной кластеризации практически невозможно. Поэтому для проектирования подобных максимально реалистичных крупномасштабных агент-ориентированных моделей появляется необходимость в использовании параллельных вычислений и суперкомпьютерных технологий. Среди множества исследований по данной тематике следует отметить работы отечественных и зарубежных ученых, в частности работы В. Л. Макарова, А. Р. Бахтизина и др. [4, 6—10, 16, 26], посвященные применению суперкомпьютерных технологий в общественных науках и, в частности, для агент-ориентированного моделирования (АОМ).

Несмотря на то что уже существуют несколько известных платформ для программной реализации подобных моделей и их запуска на суперкомпьютерах, в частности Repast HPC [14, 15], FLAME [18] и др., по-прежнему актуальной остается задача разработки отечественных аналогов подобных систем. Необходимость таких платформ связана с обеспечением процедуры быстрой разработки и отладки агентных моделей на обычных персональных компьютерах с последующим запуском реализующих их программ на высокопроизводительных вычислительных кластерах. Отметим, что платформа Repast HPC функционирует под управлением операционной среды UNIX (Linux Ubuntu) и не имеет удобных средств отладки и тестирования моделей. Программная платформа

MASON [24] основана на кроссплатформенном языке Java, имеющем ограниченные возможности в использовании технологий параллельных вычислений. Система FLAME [18] основана на технологии CUDA и ориентирована преимущественно на создание крупномасштабных моделей, требующих визуализации динамики агентов в режиме реального времени.

Цель настоящей работы — разработка модели поведения толпы в социально значимом объекте (на примере аэропорта) для минимизации "эффекта толпы". Такой эффект влияет на уровень комфорта людей и, как следствие, на скорость пассажиропотока, а также на ряд других важных факторов. Разработка модели призвана стимулировать проектирование новой программной платформы, предназначенной для крупномасштабного агент-ориентированного моделирования с использованием технологии параллельных вычислений класса MPI (*Message Passing Interface* — интерфейса передачи сообщений) и языка программирования C+.

Модель поведения толпы в аэропорту

Разрабатываемая программная платформа будет апробирована на примере модели поведения толпы в аэропорту, учитывающей одновременное обслуживание не менее 7 000 агентов. Предлагаемая в рамках настоящей публикации модель динамики пассажиропотока основана на ранее разработанных моделях поведения толпы. Она также использует феноменологический подход для описания динамики и правил поведения агентов с включением цепочки процессов (последовательностей обслуживания) и связанных с ними ресурсов (например, стоек регистрации, окон паспортного контроля и др.), влияющих на направления и плотность людских потоков.

Подобный излагаемому далее подход позволяет, в частности, учесть внутреннюю геометрию помещения и плотность толпы при выборе агентом направления движения. В результате агент либо движется к своим целевым координатам (например, к стойке регистрации, багажной ленте, зоне паспортного контроля, зоне предполетного контроля, выходу на посадку и др.), либо обходит различные препятствия, в том числе зоны сильного скопления людей.

Отметим, что предлагаемая модель переназначена в первую очередь для определения наилучшей конфигурации аэропорта, обеспечивающей максимально комфортные условия для пассажиров. Модель направлена на исключение "эффекта толпы", при котором образуются множественные локальные кластеры толпы (скопления людей) в наиболее важных зонах. К числу таких зон относятся, например, зона регистрации, зона паспортного и таможенного контроля, зона получения багажа и т. д.

При этом рассматривают две условные группы агентов — вылетающие и прилетевшие пассажиры, отличающиеся правилами поведения, которые описываются с использованием диаграммы состояний, представленной на рис. 2.

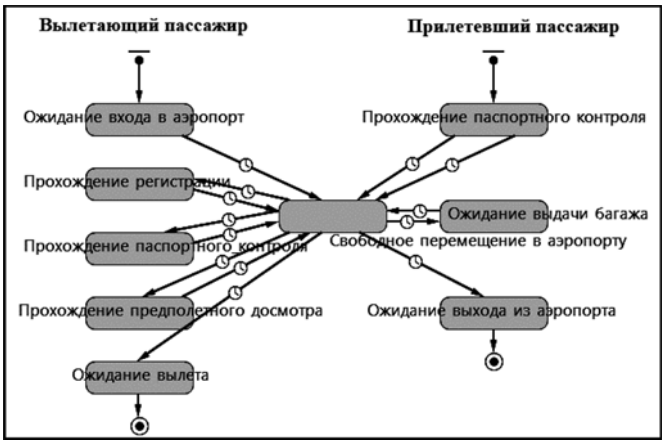


Рис. 2. Диаграмма состояний агентов-пассажиров

Отметим, что прилетевший пассажир (рис. 2) может стать вылетающим пассажиром, в случае если рейс является транзитным, т. е. предполагает пересадку. Основное состояние — "свободное перемещение в аэропорту" — используется для организации перемещения агента между зонами обслуживания, например между зоной регистрации и зоной паспортного контроля. При этом в случае нахождения агента в одном из других возможных конечных состояний (например, при прохождении предполетного досмотра, при ожидании выдачи багажа и др.) его скорость считается равной нулю (т. е. агент-пассажир неподвижен). Для каждого текущего состояния агента задаются новые целевые координаты, соответствующие релевантному центру обслуживания. Например, если текущее состояние агента "ожидание входа в аэропорт", то целевые координаты агента задаются в зоне прохождения регистрации на рейс и т. д. Предполагается также, что каждый такой центр обслуживания пассажиров имеет определенную характеристику вместимости агентов. Через вход (выход), например, может одновременно пройти несколько агентов, вдоль одной стойки регистрации могут быть расположены несколько агентов, через одну кабину паспортного контроля проходит один агент и т. д. Подобный подход позволяет смоделировать движение пассажиропотока в аэропорту в соответствии с заданными правилами поведения агентов и ресурсных характеристик рассматриваемой системы.

Перейдем к формальному описанию модели.

Введем следующие обозначения:

T — набор временных моментов (в минутах); $|T|$ — общее число временных моментов; $t_0 \in T$, $t_{|T|} \in T$ — начальные и конечные моменты времени; $t_j \in T$, $j = 0, ..., |T|$, — все моменты времени;

$I(t_j)$ — набор индексов агентов в момент t_j; $|I(t_j)|$ — общее число агентов в аэропорту в момент t_j;

$K(t_j)$ — набор индексов существующих (возникающих) препятствий (в том числе зон сильного скопления людей) в момент t_j; $|K(t_j)|$ — общее число препятствий в аэропорту, которые могут быть на пути агента в момент t_j;

$g = \{0, 1,..., 8\}$ — набор возможных состояний агентов (рис. 2): $g = 0$ — свободное перемещение в аэропорту; $g = 1$ — ожидание входа в аэропорт; $g = 2$ — прохождение регистрации (на рейс); $g = 3$ — прохождение паспортного контроля (до вылета); $g = 4$ — прохождение предполетного досмотра; $g = 5$ — ожидание вылета; $g = 6$ — прохождение паспортного контроля (по прилету); $g = 7$ — ожидание выдачи багажа; $g = 8$ — ожидание выхода из аэропорта;

$st_i(t_j) \in g$ — текущее состояние i-го агента в момент t_j, $i \in I(t_j)$;

$st_i^*(t_j) \in g$ — состояние i-го агента $i \in I(t_j)$, предшествующее переходу в состояние свободного перемещения в момент t_j;

$Z_g(t_j)$ — набор индексов функционирующих центров обслуживания пассажиров, соответствующих каждому g-му состоянию в момент t_j, например число входов (выходов) в аэропорт, число стоек регистрации, окон (кабинок) паспортного контроля и др.; $|Z_g(t_j)|$ — общее число функционирующих центров обслуживания пассажиров, релевантных g-му состоянию агентов в момент t_j ($|Z_g(t_j)|$ — управляющий параметр модели);

$\tau_i(st_i(t_j)) \in T$ — время вхождения i-го агента в состояние $st_i(t_j)$, $st_i(t_j) \in g$;

$\delta_{iz_g}\left(st_i\left(t_j\right)\right)$ — среднее время обслуживания i-го агента $i \in I(t_j)$ в z_g-м центре обслуживания; $z_g \in Z_g(t_j)$ ($\delta_{iz_g}\left(st_i\left(t_j\right)\right)$ — управляющий параметр модели);

$r_i(t_j)$ — радиус личного пространства i-го агента $i \in I(t_j)$ в момент t_j, зависящий от плотности толпы вокруг агента [11];

$\{x_i(t_j), y_i(t_j)\}$ — текущие координаты i-го агента $i \in I(t_j)$ в момент t_j;

$\left\{\tilde{x}_{iz_g}\left(t_j\right), \tilde{y}_{iz_g}\left(t_j\right)\right\}$ — целевые координаты i-го агента $i \in I(t_j)$ в момент t_j, определяемые по месту расположения z_g-го центра обслуживания $z_g \in Z_g(t_j)$;

$\left\{\hat{x}^q_{iz_g}\left(t_j\right), \hat{y}^q_{iz_g}\left(t_j\right)\right\}$ — координаты i-го агента $i \in I(t_j)$ в момент t_j, движущегося по мысленной (условной) траектории к месту расположения z_g-го центра обслуживания $z_g \in Z_g(t_j)$, $q = 1, 2,..., Q$ — индекс внутренних итераций; Q — общее число итераций (точек, принадлежащих мысленной траектории);

$\left\{\mathbf{X}_k\left(t_j\right), \mathbf{Y}_k\left(t_j\right)\right\}$ — множество координат точек, принадлежащих k-му препятствию $k \in K(t_j)$ в момент t_j;

$\left\{\tilde{\mathbf{x}}^*_{ik}\left(t_j\right), \tilde{\mathbf{y}}^*_{ik}\left(t_j\right)\right\}$ — множество возможных координат i-го агента $i \in I(t_i)$ в момент t_j, обеспечивающих обход ближайшего k-го препятствия (вычисляются посредством построения мысленного пути от i-го агента в момент t_j до центра обслуживания, и постепенного вращения спирали с центром в точке пересечения мысленной траектории с препятствием);

$\alpha_{iz_g}\left(t_j\right)$ — угол направления перемещения i-го агента $i \in I(t_j)$ к одному из z_g-х центров обслуживания пассажиров $z_g \in Z_g(t_j)$ при условии отсутствия какого-либо препятствия на пути данного агента;

$\gamma_{ik}(t_j)$ — угол направления обхода i-го агента $i \in I(t_j)$ ближайшего k-го препятствия $k \in K(t_j)$ в момент t_j (в том числе плотной толпы), находящегося на его пути и обеспечивающего минимальное отклонение от первоначальной траектории;

$\beta_{ip}(t_j)$ — угол смещения направления i-го агента $i \in I(t_j)$ в момент t_j относительно первоначального направления, обусловленного наличием ближайшего p-го агента на его пути; $p \in I(t_j)$;

$d_{ip}(t_j)$ — евклидово расстояние между i-м агентом $i \in I(t_j)$ и ближайшим p-м агентом $p \in I(t_j)$ в момент t_j;

$s_i(t_j)$ — скорость i-го агента $i \in I(t_j)$ в момент t_j, зависящая от расстояния до ближайшего агента в толпе (скорость максимальна при отсутствии других агентов).

С учетом представленных обозначений пространственная динамика i-го агента на временном интервале $t_j \leqslant \xi \leqslant t_j + 1$ описывается следующей системой дифференциальных уравнений с переменной структурой:

$$\frac{dx_i(\xi)}{d\xi} = \begin{cases} s_i(\xi)\cos\alpha_{iz_g}(\xi), & \text{если выполняется I,} \\ s_i(\xi)\cos\left(\alpha_{iz_g}(\xi) \pm \beta_{ip}(\xi)\right), & \text{если выполняется II,} \\ s_i(\xi)\cos\gamma_{ik}(\xi), & \text{если выполняется III,} \\ s_i(\xi)\cos\left(\gamma_{ik}(\xi) \pm \beta_{ip}(\xi)\right), & \text{если выполняется IV,} \\ 0, & \text{если выполняется V,} \end{cases} \quad (1)$$

$$\frac{dy_i(\xi)}{d\xi} = \begin{cases} s_i(\xi)\sin\alpha_{iz_g}(\xi), & \text{если выполняется I,} \\ s_i(\xi)\sin\left(\alpha_{iz_g}(\xi) \pm \beta_{ip}(\xi)\right), & \text{если выполняется II,} \\ s_i(\xi)\sin\gamma_{ik}(\xi), & \text{если выполняется III,} \\ s_i(\xi)\sin\left(\gamma_{ik}(\xi) \pm \beta_{ip}(\xi)\right), & \text{если выполняется IV,} \\ 0, & \text{если выполняется V,} \end{cases} \quad (2)$$

$i \in I(\xi)$, $p \in I(\xi)$, $z_g \in Z_g(\xi)$, $g = \{0, 1,..., 8\}$, $k \in K(\xi)$,

где

I. $d_{ip}(\xi) > (r_i(\xi) + r_p(\xi))$ для всех $p \in I(\xi)$ и $\left(\hat{x}^q_{iz_g}(\xi) \notin \mathbf{X}_k(\xi)$ и $\hat{y}^q_{iz_g}(\xi) \notin \mathbf{Y}_k(\xi)\right)$ для всех $q = 1, 2,..., Q$ и $st_i(\xi) = 0$,

II. $d_{ij}(\xi) \leqslant (r_i(\xi) + r_p(\xi))$ для ближайшего $p \in I(\xi)$ и $\left(\hat{x}^q_{iz_g}(\xi) \notin \mathbf{X}_k(\xi)$ и $\hat{y}^q_{iz_g}(t) \notin \mathbf{Y}_k(\xi)\right)$ для всех $q = 1, 2,..., Q$ и $st_i(\xi) = 0$,

III. $d_{ip}(\xi) > (r_i(\xi) + r_p(\xi))$ для всех $p \in I(\xi)$ и $\left(\hat{x}^q_{iz_g}(\xi) \in \mathbf{X}_k(\xi)$ и $\hat{y}^q_{iz_g}(\xi) \in \mathbf{Y}_k(\xi)\right)$ хотя бы для одного $q = 1, 2,..., Q$ и $st_i(\xi) = 0$,

IV. $d_{ip}(\xi) \leqslant (r_i(\xi) + r_p(\xi))$ для ближайшего $p \in I(\xi)$ и $\left(\hat{x}^q_{iz_g}(\xi) \in \mathbf{X}_k(\xi)$ и $\hat{y}^q_{iz_g}(\xi) \in \mathbf{Y}_k(\xi)\right)$ хотя бы для одного $q = 1, 2,..., Q$ и $st_i(\xi) = 0$,

V. $d_{ij}(\xi) \leqslant (r_i(\xi) + r_p(\xi))$ для всех $p \in I(\xi)$ или $\left(\hat{x}^q_{iz_g}(t) \in \mathbf{X}_k(t)$ и $\hat{y}^q_{iz_g}(t) \in \mathbf{Y}_k(t)\right)$ для всех $q = 1, 2,..., Q$ или $st_i(\xi) \neq 0$.

Координаты i-го агента, движущегося по мысленной (виртуальной) траектории, состоящей из точек $q = 1, 2,..., Q$, к целевому центру обслуживания в момент времени ξ:

$$\hat{x}^q_{iz_g}(\xi) = \hat{x}^{q-1}_{iz_g}(\xi) + s_i(\xi)\cos\alpha_{iz_g}(\xi), \quad (3)$$

$$\hat{y}^q_{iz_g}(\xi) = \hat{y}^{q-1}_{iz_g}(\xi) + s_i(\xi)\sin\alpha_{iz_g}(\xi), \quad (4)$$

$$i \in I(\xi), \quad z_g \in Z_g(\xi).$$

Угол направления (свободного) перемещения i-го агента $i \in I(\xi)$ к z_g-му центру обслуживания $z_g \in Z_g(\xi)$ в момент времени ξ (рис. 3):

$$\alpha_{iz_g}(\xi) = \arctan\frac{\tilde{y}_{iz_g}(\xi) - y_i(\xi)}{\tilde{x}_{iz_g}(\xi) - x_i(\xi)}, \quad (5)$$

$$i \in I(\xi), \quad z_g \in Z_g(\xi).$$

Угол смещения i-го агента $i \in I(\xi)$ в целях обхода ближайшего p-го соседа $p \in I(\xi)$, расположенного на пути его следования к центру обслуживания в момент времени ξ (рис. 4):

$$\beta_{ip}(\xi) = \frac{\pi}{4} + \arctan\frac{y_p(\xi) + \left(r_i(\xi) + r_p(\xi)\right)\sin(\pi/4) - y_i(\xi)}{x_p(\xi) + \left(r_i(\xi) + r_p(\xi)\right)\cos(\pi/4) - x_i(\xi)}, \quad (6)$$

$$i \in I(\xi), \; p \in I(\xi).$$

Множество углов возможных направлений обхода i-м агентом $i \in I(\xi)$ ближайшего k-го препятствия

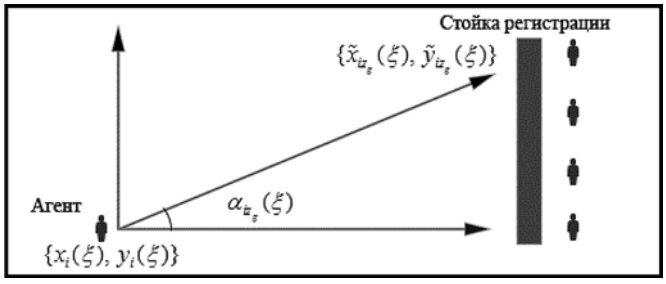


Рис. 3. Свободное перемещение агента к центру обслуживания

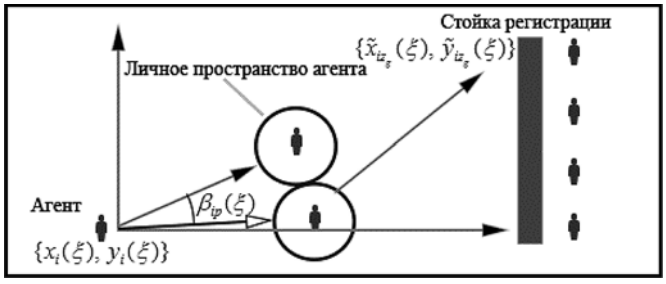


Рис. 4. Смещение агента в целях обхода ближайшего соседа при перемещении к центру обслуживания

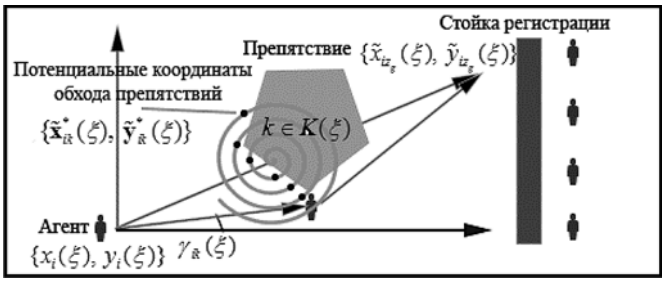


Рис. 5. Обход агентом ближайшего препятствия при перемещении к центру обслуживания

$k \in K(\xi)$, расположенного на пути его следования к z_g-му центру обслуживания $z_g \in Z_g(\xi)$ в момент времени ξ (рис. 5):

$$\tilde{\boldsymbol{\gamma}}_{ik}(\xi) = \arctan\frac{\tilde{\mathbf{y}}^*_{ik}(\xi) - y_i(\xi)}{\tilde{\mathbf{x}}^*_{ik}(\xi) - x_i(\xi)}, \quad (7)$$

$$i \in I(\xi), \; k \in K(\xi).$$

Угол направления обхода i-м агентом $p \in I(\xi)$ ближайшего k-го препятствия $k \in K(\xi)$, расположенного на его пути, обеспечивающий минимальное отклонение от первоначальной траектории в момент времени ξ:

$$\gamma_{ik}(\xi) = \arg\min\left|\alpha_{iz_g}(\xi) - \tilde{\boldsymbol{\gamma}}_{ik}(\xi)\right|, \quad (8)$$

$$i \in I(\xi), \quad z_g \in Z_g(\xi), \quad g = \{0, 1,..., 8\}, \quad k \in \tilde{K}(\xi).$$

Отметим, что известны и применяются многие другие алгоритмы обхода препятствий, например алгоритм Дейкстры, алгоритм "A star" и др. [19]. Однако они характеризуются большей вычислительной сложностью и трудно применимы для крупномасштабных агент-ориентированных моделей. Кроме того, предполагается, что в социально-ориентированных объектах (например, аэропорт) маршрутизация агентов поддерживается с помощью множественных информационных указателей, исключающих необходимость "слепого" поиска центров обслуживания. При этом возникновение более простых локальных препятствий (в том числе плотных скоплений людей) является типичным явлением для подобных систем.

Важной характеристикой предложенной модели являются правила перехода i-го агента к новым состояниям $st_i(t_j) \in g$:

$$st_i(t) = \begin{cases} 0, \text{ если } t_j = t_0 \\ \text{или } t_j \geqslant \tau_i\left(st^*_i(t_j)\right) + \delta_{iz_g}\left(st^*_i(t_j)\right), \; st^*_i(t_j) \in g, \\ \{1, 2, \; ..., \; 8\}, \text{ если } x_i(t) \in \left\{\tilde{x}_{iz_g}(t_j), \; \tilde{y}_{iz_g}(t_j)\right\} \\ \text{или } t_j < \tau_i\left(st^*_i(t_j)\right) + \delta_{iz_g}\left(st^*_i(t_j)\right), \end{cases} \quad (9)$$

$$i \in I(t_j), \quad z_g \in Z_g(t_j), \quad g = \{0, 1,..., 8\}.$$

Приведенные правила обеспечивают переход i-го агента $i \in I(t_j)$ к новым состояниям, например, состоянию свободного перемещения в аэропорту $st_i(t_j) = 0$, предполагающему высвобождение данного агента из z_g-го центра обслуживания ($z_g \in Z_g(t_j)$) и назначения ему новых целевых координат $\left\{\tilde{x}_{iz_g}(t_j),\ \tilde{y}_{iz_g}(t_j)\right\}$. При достижении агентом этих целевых координат он переходит в новое состояние $st_i(t_j) \in \{1, 2,..., 8\}$ и остается в нем до окончания времени обслуживания.

Для оценки плотности толпы в аэропорту внутреннее пространство аэропорта делится на M квадратных областей равномерно распределенными горизонтальными и вертикальными прямыми. При этом число клеток является параметром моделирования. Каждый подобный квадрат может быть описан окружностью с фиксированным радиусом R_m, $m = 1, 2,..., M$. Далее оценивается число агентов, находящихся в каждой пространственной области.

Таким образом, плотность агентов в m-й области, $m = 1, 2,..., M$, в каждый момент времени $t_j \in T$ равно

$$\rho = \frac{1}{\pi R_m^2} \sum_{i=1}^{|I(t)|} \psi_{im}(t_j), \tag{10}$$

$$\psi_{im}(t_j) = \begin{cases} 1, \text{ если } \tilde{d}_{im}(t_j) \leqslant R_m, \\ 0, \text{ если } \tilde{d}_{im}(t_j) > R_m, \end{cases} \tag{11}$$

где $\psi_{im}(t_j)$ — характеристическая функция присутствия i-го агента $i \in I(t_j)$ в m-й области ($m = 1, 2,..., M$) в момент времени t_j;

$\tilde{d}_{im}(t_j)$ — евклидово расстояние i-го агента от центра m-й области ($m = 1, 2, ..., M$) в момент времени t_j.

Если плотность агентов в m-й области превышает некоторый пороговый уровень (например, более 1 агента на 2 м^2), то будем считать, что в данной области высокая плотность толпы.

Далее рассмотрим разработанную целевую программную платформу, которая обеспечивает реализацию крупномасштабной агентной модели поведения толпы в аэропорту. Более подробно она представлена в работах [1—11]. Такая модель учитывает влияние различных факторов, например число входов и выходов, число стоек регистрации, геометрию помещения, число окон паспортного контроля, время ожидания багажа и ряд других.

Описание разработанной программной платформы

Для компьютерной реализации предложенной имитационной модели была спроектирована новая программная платформа с использованием среды C++, технологии MPI и библиотеки классов boost, которые существенно упрощают процедуру распараллеливания агентной модели. Подобный стек технологий был выбран как один из наиболее эффективных инструментальных средств реализации крупномасштабных агент-ориентированных моделей.

Отметим, что до реализации данной модели в среде C++ ее малоразмерный прототип был создан в системе AnyLogic [1], что позволило протестировать общие функциональные возможности системы, в особенности в части механизма формирования и диссипации кластеров толпы.

Укрупненная архитектура разработанной с участием авторов программной платформы для реализации крупномасштабных АОМ представлена на рис. 6. Отметим, что существенное повышение производительности по сравнению с традиционными системами агент-ориентированного имитационного моделирования обеспечивается за счет использования технологии MPI [17], позволяющей, в частности, равномерно распределить агентов по процессам. В результате сложные внутренние вычислительные процедуры и правила поведения каждого агента во взаимодействии с другими агентами реализуются для ограниченного числа агентов, принадлежащих данному процессу. При этом характеристики агентов (например, их координаты в пространстве, состояние и др.), принадлежащих другим процессам, являются известными, в частности, посредством синхронизации, обеспечиваемой функцией **all_gather().**

Использование векторов для хранения агентов позволяет легко организовать процедуру создания и добавления новых агентов в популяцию, например, с использованием следующего оператора:

myAgentList.push_back(Agent_p(world.rank(), i, [...] // Параметры Агента)).

Соответственно, возможно и удаление агента из популяции с использованием функции **erase()**. Важный элемент спроектированной программной платформы — это класс агента **Agent_p,** являющийся родительским (базовым) классом для всех агентов, которые могут быть в дальнейшем динамически созданы в модели. Данный класс содержит ключевые характеристики, присущие всем типам агентов (например, уникальный идентификатор процесса, пространственные координаты, состояние агента и др.). При этом в среде C++ можно легко создать производный класс, наследующий все характеристики базового класса с индивидуальным набором параметров и операций, например, посредством оператора **class** Person: **public** Agent_p{}, с помощью которого будет создан дочерний агентный класс **Person** (человек).

Важным преимуществом спроектированной программной платформы является возможность реализации как простых, так и сложных взаимодействий между агентами. Так как в каждый момент модельного времени каждый агент обладает всей полнотой информации о состоянии всех других агентов (рис. 6), это позволяет задать любые правила их взаимодействия. При этом для повышения временной эффективности подобной системы необходимо ограничивать размерности популяций взаимодействующих агентов по определенному критерию.

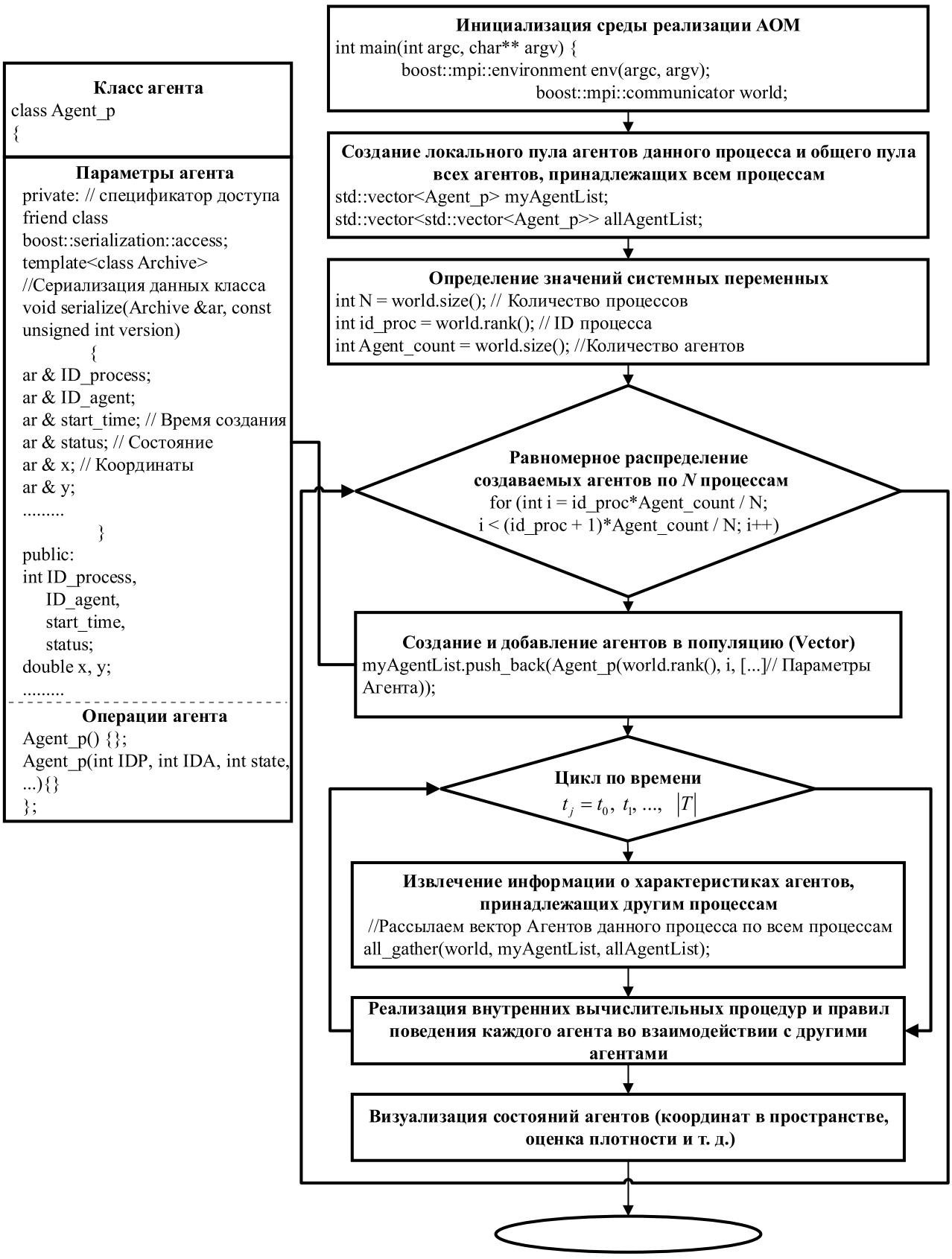


Рис. 6. Укрупненная архитектура разработанной программной платформы реализации крупномасштабных АОМ

Например, следует учитывать только близкорасположенных агентов либо агентов, между которыми могут возникнуть коммуникационные связи. Таким образом, спроектированная программная платформа может быть дополнена модулем кластеризации агентов в целях снижения размерности решаемой задачи. При моделировании поведения толпы в аэропорту всю популяцию агентов предварительно целесообразно разбить на кластеры в зависимости от их пространственного расположения. В результате при описании возможных столкновений между агентами рассматриваются только агенты, принадлежащие одному кластеру.

Заметим, что созданная платформа позволяет интегрировать программно реализуемые агентные модели с различными базами данных, оптимизационными алгоритмами и внешними библиотеками, в том числе написанными на других языках программирования. Подобная платформа является достаточно универсальной и может быть использована для компьютерной реализации различных агентных моделей. Важным преимуществом разработанной программной платформы является возможность ее реализации для различных операционных систем как семейства Windows, так и UNIX (в отличие от платформы Repast HPC [14, 15], которая функционирует только в среде UNIX). В результате обеспечивается удобная процедура разработки, конфигурирования и отладки моделей, в частности, с использованием MS Visual Studio с установленными библиотеками **boost**. Также важным преимуществом разработанной программной платформы в сравнении с другими известными системами агентного моделирования (например, AnyLogic, NetLogo, MASON, Mesa и др.) является существенно лучшее быстродействие за счет использования языка программирования C++ и технологии параллельных вычислений MPI.

Результаты имитационного моделирования

Отметим, что вычислительные эксперименты проводились на примере аэропорта Домодедово, для которого ранее в работе [13] были исследованы различные сценарии эвакуации агентов при возникновении чрезвычайной ситуации. Эксперименты проводили с использованием разработанной программной платформы (рис. 6), позволяющей, в частности, распараллелить все вычислительные процедуры, относящиеся к каждому агенту. На рис. 7 представлено распределение агентов для текущей конфигурации внутреннего пространства аэропорта Домодедово, базовые характеристики которого представлены в таблице.

Заметим, что имеются и другие параметры, учитываемые при моделировании динамики пассажиропотока в аэропорту. К их числу относятся: время ожидания входа в аэропорт и прохождения рамок металлодетектора; число и вместимость залов ожидания; число бизнес-залов; число сидячих мест и др.

Первая серия экспериментов была нацелена на определение областей формирования кластеров толпы

Базовые характеристики аэропорта Домодедово, влияющие на динамику пассажиропотока (по данным на 2018 г.)

Показатель	Значение
Средний пассажиропоток в сутки	60 000 чел.
Среднее число одновременно находящихся людей (в том числе пассажиров, сопровождающих лиц, сотрудников и др.)	7000 чел.
Общая площадь всех терминалов	135 тыс. $м^2$
Число взлетно-посадочных полос (ВПП)	2
Пропускная способность ВПП (среднее число посадок и приземлений в час)	80
Среднее число пассажиров на 1 рейс	200 чел.
Число стоек регистрации	130
Пропускная способность стойки регистрации	80 чел/ч
Число стоек самостоятельной регистрации	24
Пропускная способность стойки самостоятельной регистрации	20 чел/ч
Число окон (кабин) паспортного контроля	128
Пропускная способность окна паспортного контроля	10 чел/ч
Число досмотровых линий в зоне предполетного контроля	15
Пропускная способность досмотровых линий	360 чел/ч
Среднее время ожидания получения багажа	40 мин
Число входов в аэропорт	4

(зон скопления людей), влияющих на среднее время пребывания агента-пассажира в аэропорту (рис. 7).

На рис. 7 черным цветом выделены агенты, относящиеся к кластерам толпы с высокой плотностью (менее 1 чел. на 2 $м^2$), а серым цветом — все остальные агенты.

Как видно на рис. 7, наибольшие очереди и скопления агентов образуются при входе в аэропорт, при регистрации на рейсы, в зоне паспортного и таможенного контроля. Наблюдается также высокая плотность толпы в секторе зоны выдачи багажа.

Далее была проведена серия экспериментов с варьированием ключевых характеристик аэропорта (см. таблицу), влияющих на характеристики кластеров толпы. Отметим, что существенным фактором, влияющим на пассажиропоток в аэропорту, является число действующих ВПП, строительство и ввод в эксплуатацию которых представляет собой сложную, ресурсозатратную задачу. Так, например, увеличение числа ВПП с 2 до 4 запланировано только к 2037 г. По этой причине для вариационных экспериментов были выбраны ресурсные характе-

ристики, изменение которых требует наименьших затрат, в частности, следующие:

- число входов в аэропорт;
- число стоек регистрации;
- число стоек самостоятельной регистрации;
- число действующих окон паспортного контроля;
- число досмотровых линий в зоне предполетного контроля;
- среднее время ожидания багажа.

Значения остальных параметров были фиксированы.

В результате имитационного моделирования были найдены рациональные значения соответствующих управляющих параметров, определяющих внутреннюю конфигурацию аэропорта, при которых обеспечивается максимально возможный уровень пропускной способности пассажиропотока в аэропорту (рис. 8).

Как видно на рис. 8, основными факторами, влияющими на формирование кластеров толпы в аэропорту Домодедово, являются число входов, число действующих окон паспортного контроля и среднее время ожидания получения багажа. Таким образом, необходимо: строительство еще одного входа в здание аэропорта; двукратное увеличение числа окон паспортного контроля и, как следствие, снижение среднего времени ожидания багажа до 30 мин. Открытие дополнительных 20 стоек регистрации приведет к фактическому устранению (диссипации) кластеров толпы, сокращению очередей и, таким образом, к повышению уровня комфорта пассажиров.

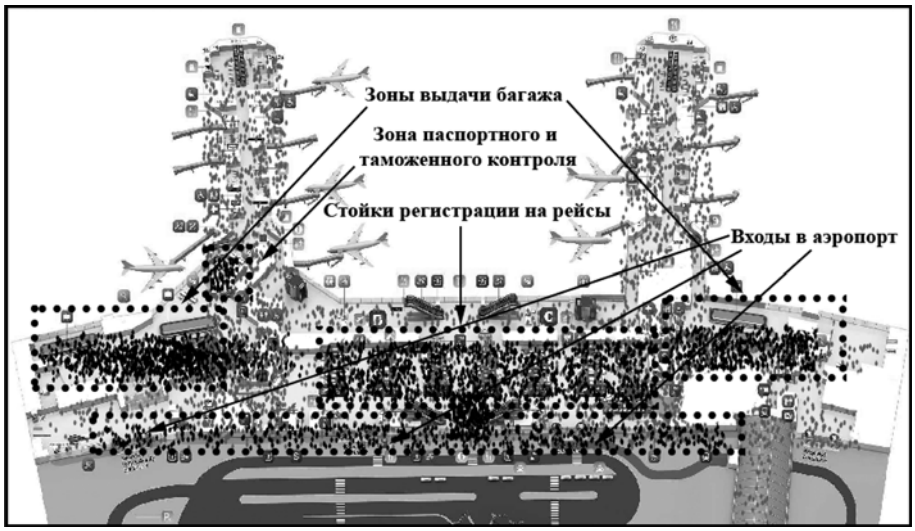


Рис. 7. Распределение людских потоков и формирование кластеров толпы при текущей конфигурации внутреннего пространства аэропорта

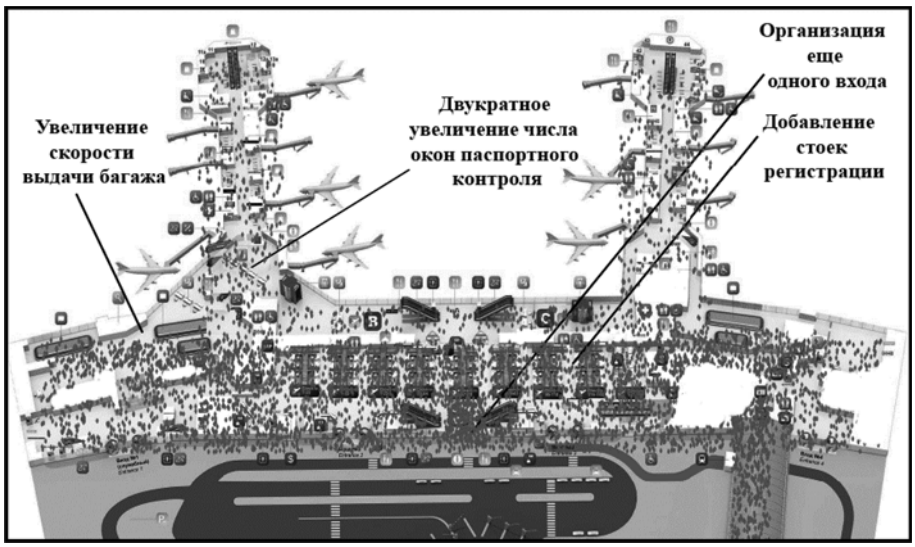


Рис. 8. Распределение людских потоков и формирование кластеров толпы при новой (рациональной) конфигурации внутреннего пространства аэропорта

Заключение

В настоящей работе представлен пример реализации крупномасштабной агентной модели поведения толпы в аэропорту, учитывающей влияние различных факторов. Подобная модель основана на ранее предложенном феноменологическом подходе [2, 11], дополненном индивидуальной системой принятия решений агента с учетом состояния окружающего его пространства, например наличия препятствий, возникновения плотной толпы на пути агента и др. (см. рис. 1).

Впервые разработана крупномасштабная модель поведения толпы в аэропорту, особенностью которой является включение цепочки процессов (последовательностей обслуживания) и связанных с ними ресурсов (например, стоек регистрации, окон паспортного контроля и др.), влияющих на направления и плотность людских потоков (см. рис. 2). Пространственная динамика агентов описывается системой дифференциальных уравнений с переменной структурой (1)—(2), учитывающей различные сценарии взаимодействия агентов друг с другом и с более сложными препятствиями (см. рис. 3—5).

Спроектирована оригинальная программная платформа реализации крупномасштабных агент-ориентированных моделей с использованием технологии C++ и MPI (см. рис. 6). Важной особенностью данной платформы является возможность эффективного распараллеливания агентных моделей,

использования механизма наследования для создания агентов различного типа, динамического создания и удаления агентов из популяции и др. Подобная платформа является достаточно универсальной и может быть использована для компьютерной реализации различных агентных моделей.

С использованием созданной системы на примере аэропорта Домодедово проведены численные эксперименты. В результате продемонстрировано (см. рис. 7, 8), что организация еще одного входа в здание аэропорта, двукратное увеличение числа окон паспортного контроля, снижение среднего времени ожидания багажа до 30 мин и открытие дополнительных 20 стоек регистрации приведет к фактическому устранению кластеров толпы (минимизации плотности толпы), что повысит уровень комфортности авиаперевозок.

Список литературы

1. **Акопов А. С.** Имитационное моделирование: учебник и практикум для академического бакалавриата. М.: Юрайт, 2015. 389 с.
2. **Акопов А. С., Бекларян Л. А.** Агентная модель поведения толпы при чрезвычайных ситуациях // Автоматика и телемеханика. 2015. № 10. С. 131—143.
3. **Антонов А. С.** Параллельное программирование с использованием технологии MPI: Учебное пособие. М.: Изд-во МГУ, 2004. 71 с.
4. **Бахтизин А. Р.** Агент-ориентированные модели экономики. М.: Экономика, 2008. 279 с.
5. **Бекларян А. Л., Акопов А. С.** Моделирование поведения толпы на основе интеллектуальной динамики взаимодействующих агентов // Бизнес-информатика. 2015. № 1 (31). С. 69—77.
6. **Макаров В. Л., Бахтизин А. Р., Сушко Е. Д., Сушко Г. Б.** Моделирование социальных процессов на суперкомпьютерах: новые технологии // Вестник Российской академии наук. 2018. Т. 88, № 6. С. 508—518.
7. **Макаров В. Л., Бахтизин А. Р., Сушко Е. Д., Васенин В. А., Борисов В. А., Роганов В. А.** Агент-ориентированные модели: мировой опыт и технические возможности реализации на суперкомпьютерах // Вестник Российской академии наук. 2016. Т. 86, № 3. С. 252—262.
8. **Макаров В. Л., Бахтизин А. Р., Сушко Е. Д., Васенин В. А., Борисов В. А., Роганов В. А.** Суперкомпьютерные технологии в общественных науках: агент-ориентированные демографические модели // Вестник Российской академии наук. 2016. Т. 86, № 5. С. 412—421.
9. **Макаров В. Л., Бахтизин А. Р.** Применение суперкомпьютерных технологий в общественных науках // Экономика и математические методы. 2013. Т. 49, № 4. С. 18—32.
10. **Макаров В. Л., Бахтизин А. Р.** Социальное моделирование — новый компьютерный прорыв (агент-ориентированные модели). М.: Экономика, 2013. 295 с.
11. **Akopov A. S., Beklaryan L. A.** Simulation of human crowd behavior in extreme situations // International Journal of Pure and Applied Mathematics. 2012. Vol. 79, No. 1. P. 121—138.
12. **Antonini G., Bierlaire M., Weber M.** Discrete choice models of pedestrian walking behavior // Transportation Research Part B. 2006. Vol. 40, No. 8. P. 667—687.
13. **Beklaryan A. L., Akopov A. S.** Simulation of Agent-rescuer Behaviour in Emergency Based on Modified Fuzzy Clustering// AAMAS'16: Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems. Richland: International Foundation for Autonomous Agents and Multiagent Systems, 2016. P. 1275—1276.
14. **Collier N., North M.** Repast HPC: A Platform for Largescale Agent-based Modeling // Large-Scale Computing Techniques for Complex System Simulations, Wiley, 2011. P. 81—110.
15. **Collier N.** Repast HPC Manual, 13th August, 2013. URL: https://repast.github.io/docs/repast_hpc.pdf (дата обращения 24.12.2018).
16. **Gilbert N., Troitzsch K. G.** Simulation for the Social Scientist. Open University Press, 2nd ed., 2005. 295 p.
17. **Gregor D. Troyer M.** Boost.MPI. 2005. URL: http://boost.cowic.de/rc/pdf/mpi.pdf (дата обращения 24.12.2018).
18. **Coakley S., Gheorghe M., Holcombe M., Chin S., Worth D., Greenough C.** Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework // In proceedings of IEEE 14th International Conference on High Performance Computing and Communications, 2012. P. 538—545.
19. **Hart P. E., Nilsson N. J., Raphael B. A.** Formal Basis for the Heuristic Determination of Minimum Cost Paths // IEEE Transactions on Systems Science and Cybernetics SSC4. 1968. No 2. P. 100—107.
20. **Helbing D., Molnar P.** Social force model for pedestrian dynamics // Physical review E. 1995. Vol. 51, No. 5. P. 4282—4286.
21. **Helbing D., Johansson A**. Al-Abideen H. Z. Dynamics of crowd disasters: An empirical study // Physical review E. 2007. Vol. 75, No. 4. P. 0461091—0461097.
22. **Helbing D., Johansson A., Al-Abideen H. Z.** Crowd turbulence: The physics of crowd disasters // The Fifth International Conference on Nonlinear Mechanics (ICNM-V), Shanghai, 2007. P. 967—969.
23. **Johansson A., Helbing D., Al-Abideen H. Z., Al-Bosta S.** From crowd dynamics to crowd safety: A video-based analysis // Advances in Complex Systems. 2008. Vol. 11, No. 4. P. 497—527.
24. **Luke S., Cioffi-Revilla C., Panait L., Sullivan K., Balan G.** MASON: A multi-agent simulation environment // Simulation: Transactions of the society for Modeling and Simulation International. 2005. Vol. 82, No. 7. P. 517—527.
25. **Mintz A.** Non-adaptive group behavior // Journal of Abnormal Psychology. 1951. Vol. 46, No. 2. P. 150—159.
26. **Rosset L. M., Nardin L. G., Sichman J. S.** Use of High Performance Computing in Agent-Based Social Simulation: A Case Study on Trust-Based Coalition Formation // 7th Workshop-School on Agent Systems, Environments and Applications, 2013. P. 1—3.

Development of Software Framework for Large-Scale Agent-Based Modeling of Complex Social Systems

V. L. Makarov[1], makarov@cemi.rssi.ru, **A. R. Bakhtizin**[1], albert.bakhtizin@gmail.com,
G. L. Beklaryan[1], lbeklaryan@gmail.com, **A. S. Akopov**[1, 2], akopovas@umail.ru
[1] CEMI RAS,
[2] National Research University Higher School of Economics, Moscow, Russian Federation

Corresponding author:

Akopov Andranik S., Professor, Chief Researcher, CEMI RAS, Professor, National Research University Higher School of Economics, Moscow, 101000, Russian Federation,
E-mail: akopovas@umail.ru

Received on December 24, 2018
Accepted on January 14, 2019

In this paper, a novel approach to designing software framework that is intended for large-scale simulation modelling of complex social systems is presented. Such systems require applying supercomputing and parallel computations technologies, as well as phenomenological descriptions of agent behavior and their interactions. An agent model of human crowd behavior in an airport has been developed, taking into account the influence of different factors, for example, the number of entries and exits, the number of check-in counters, the number passport control offices, the time of waiting for baggage and other characteristics. It allows defining the best values of important resource parameters of an airport that provide a dissipation of crowd clusters. A framework based on using parallel computations MPI (Message Passing Interface), programming language C++ and the boost library is designed. The feature of such approach is the comfortable procedure of developing, configuring and debugging models. At the same time, all main advantages of earlier known agent modelling systems (e.g. AnyLogic, NetLogo, Repast HPC, FLAME, etc.) are kept and provided with the possibility of using supercomputing technologies. Some simulation experiments are conducted and they demonstrate the possibility of dissipating human crowd clusters in the airport when improving some resource characteristics, e.g. increasing of check-in counters, adding new entry, etc. In the result, the efficiency of developed software framework for the implementation of large agent models of complex social systems is shown.

Keywords: *agent-based modeling, social systems, parallel computations, crowd model, supercomputing technologies, C++, MPI, software framework for simulation modelling*

For citation:

Makarov V. L., Bakhtizin A. R., Beklaryan G. L., Akopov A. S. Development of Software Framework for Large-Scale Agent-Based Modeling of Complex Social Systems, *Programmnaya Ingeneria,* 2019, vol. 10, no. 4, pp. 167—177.

DOI: 10.17587/prin.10.167-177

References

1. **Akopov A. S.** *Imitacionnoe modelirovanie. Uchebnik i praktikum* (Simulation modeling. Textbook and workshop), Moscow, YURAJT, 2015, 389 p. (in Russian).

2. **Akopov A. S., Beklaryan L. A.** An Agent Model of Crowd Behavior in Emergencies, *Automation and Remote Control,* 2015, no. 10, pp. 1817—1827.

3. **Antonov A. S.** *Parallel'noe programmirovanie s ispol'zovaniem tekhnologii MPI: Uchebnoe posobie* (Parallel programming using MPI technology: a Training manual), Moscow, Izdatel'stvo MGU, 2004, 71 p. (in Russian).

4. **Bakhtizin A. R.** *Agent-orientirovannye modeli ehkonomiki* (Agent-based models of the economy), Moscow, Ekonomika, 2008, 279 p. (in Russian).

5. **Beklaryan A. L., Akopov A. S.** Modelirovanie povedeniya tolpy na osnove intellektual'noj dinamiki vzaimodejstvuyushchih agentov (Simulation of human crowd behavior based on intellectual dynamics of interacting agents), *Business Informatics,* 2015, no. 1 (31), pp. 69—77 (in Russian).

6. **Makarov V. L., Bakhtizin A. R., Sushko E. D., Sushko G. B.** Modelirovanie social'nyh processov na superkomp'yuterah: novye tekhnologii (Modeling of social processes on supercomputers: new technologies), *Vestnik Rossijskoj akademii nauk,* 2018, vol. 88, no. 6, pp. 508—518 (in Russian).

7. **Makarov V. L., Bahtizin A. R., Sushko E. D., Vasenin V. A., Borisov V. A., Roganov V. A.** Agent-orientirovannye modeli: mirovoj opyt i tekhnicheskie vozmozhnosti realizacii na superkomp'yuterah (Agent-based models: world experience and technical possibilities of implementation on supercomputers), *Vestnik Rossijskoj akademii nauk,* 2016, vol. 86, no. 3, pp. 252—262 (in Russian).

8. **Makarov V. L., Bahtizin A. R., Sushko E. D., Vasenin V. A., Borisov V. A., Roganov V. A.** Superkomp'yuternye tekhnologii v obshchestvennyh naukah: agent-orientirovannye demograficheskie modeli (Supercomputer technologies in the social Sciences: an agent-based demographic model), *Vestnik Rossijskoj akademii nauk,* 2016, vol. 86, no. 5, pp. 412—421 (in Russian).

9. **Makarov V. L., Bahtizin A. R.** Primenenie superkomp'yuternyh tekhnologij v obshchestvennyh naukah (Application of supercomputer technologies in social Sciences), *Ekonomika i matematicheskie metody*, 2013, vol. 49, no. 4, pp. 18—32 (in Russian).

10. **Makarov V. L., Bahtizin A. R.** *Social'noe modelirovanie — novyj komp'yuternyj proryv (agent-orientirovannye modeli)* (Social modeling is a new computer breakthrough (agent-oriented models)), Moscow, Ekonomika, 2013, 295 p. (in Russian).

11. **Akopov A. S., Beklaryan L. A.** Simulation of human crowd behavior in extreme situations, *International Journal of Pure and Applied Mathematics*, 2012, vol. 79, no. 1, pp. 121—138.

12. **Antonini G., Bierlaire M., Weber M.** Discrete choice models of pedestrian walking behavior, *Transportation Research Part B,* 2006, vol. 40, no. 8, pp. 667—687.

13. **Beklaryan A. L., Akopov A. S.** Simulation of Agent-rescuer Behaviour in Emergency Based on Modified Fuzzy Clustering, *AAMAS'16: Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, Richland: International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 1275—1276.

14. **Collier N., North M.** Repast HPC: A Platform for Large-scale Agent-based Modeling, *Large-Scale Computing Techniques for Complex System Simulations*, Wiley, 2011, pp. 81—110.

15. **Collier N.** Repast HPC Manual, 13th August, 2013, available at: https://repast.github.io/docs/repast_hpc.pdf

16. **Gilbert N., Troitzsch K. G.** *Simulation for the Social Scientist,* Open University Press, 2nd ed., 2005, 295 p.

17. **Gregor D. Troyer M.** Boost.MPI. 2005, available at: http://boost.cowic.de/rc/pdf/mpi.pdf.

18. **Coakley S., Gheorghe M., Holcombe M., Chin S., Worth D., Greenough C.** Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework, *In proceedings of IEEE 14th International Conference on High Performance Computing and Communications*, 2012, pp. 538—545.

19. **Hart P. E., Nilsson, N. J., Raphael B. A.** Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics SSC4*, 1968, no. 2, pp. 100—107.

20. **Helbing D., Molnar P.** Social force model for pedestrian dynamics, *Physical review E,* 1995, vol. 51, no. 5, pp. 4282—4286.

21. **Helbing D., Johansson A**. Al-Abideen H. Z. Dynamics of crowd disasters: An empirical study, *Physical review E,* 2007, vol. 75, no. 4, pp. 0461091—0461097.

22. **Helbing D., Johansson A., Al-Abideen H. Z.** Crowd turbulence: The physics of crowd disasters, *The Fifth International Conference on Nonlinear Mechanics (ICNM-V)*, Shanghai, 2007, pp. 967—969.

23. **Johansson A., Helbing D., Al-Abideen H. Z., Al-Bosta S.** From crowd dynamics to crowd safety: A video-based analysis, *Advances in Complex Systems*, 2008, vol. 11, no. 4, pp. 497—527.

24. **Luke S., Cioffi-Revilla C., Panait L., Sullivan K. Balan G.** MASON: A multi-agent simulation environment, *Simulation: Transactions of the society for Modeling and Simulation International,* 2005, vol. 82, no. 7, pp. 517—527.

25. **Mintz A.** Non-adaptive group behavior, *Journal of Abnormal Psychology*, 1951, vol. 46, no. 2, pp. 150—159.

26. **Rosset L. M., Nardin L. G., Sichman J. S.** Use of High Performance Computing in Agent-Based Social Simulation: A Case Study on Trust-Based Coalition Formation, *7th Workshop-School on Agent Systems, Environments and Applications*, 2013, pp. 1—3.

УДК 004.051 DOI: 10.17587/prin.10.178-185

П. М. Николаев, д-р техн. наук, начальник отдела, e-mail: geom3d@gmail.com,
Федеральное государственное унитарное предприятие "Центральный аэрогидродинамический институт имени профессора Н. Е. Жуковского", г. Жуковский

Использование локальной памяти потока для расчета B-сплайнов в задачах параллельного программирования

Рассматривается способ соотнесения динамически выделенной памяти с отдельным потоком для сокращения числа блокировок в задачах параллельного программирования. Представлена программная реализация в виде (C++)-класса, обеспечивающего сохранение указателя на локальную для текущего потока память и последующий доступ к ней. На основе использования квалификатора thread_local создан механизм освобождения соотнесенной с потоком памяти при его завершении. Приведен пример применения разработанного класса при расчете значений B-сплайнов в многопоточной среде.

Ключевые слова: *параллельное программирование, многопоточный код, асинхронный доступ к данным, локальная память потока, B-сплайны*

Введение

Широкое распространение вычислительных систем с многоядерными процессорами обусловило активное развитие инструментальных средств для создания программного обеспечения с параллельно выполняющимся кодом. Современные средства разработчика позволяют создавать эффективный многопоточный код, предоставляя необходимый набор базовых строительных блоков и удобную среду отладки программ. В современные версии популярных языков программирования входит стандартная поддержка многопоточного кода. Разработчику остается только грамотным образом скомпоновать предоставленные соответствующей библиотекой объекты для воплощения в жизнь своей идеи. Но, несмотря на наличие развитых высокоуровневых инструментальных средств, написание многопоточных программ остается сравнительно сложной задачей, требующей особого подхода.

Основной трудностью, с которой сталкиваются программисты, остается асинхронный доступ к совместно используемым данным из параллельно выполняющихся потоков. Для синхронизации доступа к данным из отдельных потоков чаще всего организуют критические секции кода, основанные на применении различных блокирующих объектов. Блокировки в целом решают проблему синхронизации, но при этом имеют свою цену, внося ряд дополнительных сложностей. Прежде всего, это риск возникновения взаимных блокировок при недостаточно продуманной организации синхронизирующих объектов. Кроме этого, само наличие критических секций уменьшает степень параллельности исполняемого кода, снижая общую производительность расчетного алгоритма. Поэтому программисты стараются свести к минимуму как размер, так и число используемых критических секций.

Одним из возможных способов сокращения числа требуемых критических секций в отдельных случаях может быть применение локальной памяти потока. Суть метода заключается в организации данных так, что у каждого отдельного потока будет свой локальный экземпляр объявленных глобально данных. Таким образом, потоку уже не требуется синхронизировать доступ к своей копии данных. В операционной системе Windows для доступа к локальной памяти потока имеются функции TlsAlloc, TlsFree, TlsGetValue, TlsSetValue [1]. В настоящей работе рассматривается применение объектов со специальным квалификатором типа памяти thread_local, введенным в стандарт C++ 11 [2], для построения механизма динамического выделения локальной памяти потока. Разработанный механизм, инкапсулированный в класс TLM, может использоваться в составе других объектов и обеспечивать доступ к областям памяти, локальным по отношению к текущему потоку.

Следует отметить, что описываемый способ не решает проблему там, где нужен действительно совместный доступ к одним и тем же данным из разных потоков. Тем не менее имеется определенный круг задач, где полезно иметь отдельную копию данных для каждого выполняемого потока. Об одной из таких задач — расчете значений B-сплайна — пойдет речь далее.

B-сплайны в многопоточной среде

B-сплайны получили широкое распространение в системах автоматизированного проектирования и производства. Они используются в качестве базовых

строительных элементов для создания и представления математических моделей двумерных и пространственных объектов. Интерес к ним обусловлен прежде всего простотой и наглядностью геометрического описания при проектировании сложных форм. Для расчета значений B-сплайнов в большинстве случаев используются алгоритмы, описанные в работах [3, 4]. Эффективность данных алгоритмов с точки зрения минимизации времени вычислений обеспечивается использованием вспомогательных массивов для хранения промежуточных результатов расчета. На листинге 1 приведен пример реализации класса объектов B-сплайнов. Функциональные возможности класса сильно сокращены с целью показать только основу алгоритма расчета.

Не вдаваясь в подробности работы алгоритма, обратим внимание на использование только вспомогательных массивов left и right в функции basis_funs. В однопоточной среде функция работает без побочных эффектов. Однако целесообразно рассмотреть, что будет, если для одного объекта Bspline функция basis_funs будет вызываться одновременно из нескольких потоков. Например, как показано на листинге 2.

```
class Bspline
{
 public:
   Bspline::Bspline(const double *knots, int degree, int nmb)
      : the_p(degree), the_n(nmb - degree - 2), U(knots, knots + nmb)
   {
      left.reset(new double[degree + 1]);
      right.reset(new double[degree + 1]);
   }

   void basis_funs(int i, double u, double *b) const;

 private:
   std::vector<double> U;
   int the_n;
   int the_p;
   mutable std::unique_ptr<double[]> left;
   mutable std::unique_ptr<double[]> right;
};

void Bspline::basis_funs(int i, double u, double *b) const
{
   b[0] = 1.;
   for (int j = 1; j <= the_p; ++j)
   {
      left[j] = u - U[i + 1 - j];
      right[j] = U[i + j] - u;
      double saved = 0.;
      for (int r = 0; r < j; ++r)
      {
         double temp = b[r] / (right[r + 1] + left[j - r]);
         b[r] = saved + right[r + 1] * temp;
         saved = left[j - r] * temp;
      }
      b[j] = saved;
   }
}
```

Листинг 1. Реализация класса B-сплайнов

```
extern Bspline bspl;

void Calc()
{
   const int N = 100;
   const int the_p = 3;
   double b[N][the_p+1];

   #pragma omp parallel for
   for(int i = 0; i < N; ++i)
      bspl.basis_funs(0, 1. / (i + 1), b[i]);
}
```

Листинг 2. Расчет значений B-сплайнов в многопоточной среде

Директива "omp parallel for" предписывает отдельным итерациям следующего за ней цикла выполняться в параллельном режиме при наличии нескольких вычислительных единиц (процессоров, ядер) [5]. Сразу видно, что совместное использование массивов left и right из разных потоков в объекте bspl приведет к повреждению сохраненных в них промежуточных результатов расчета.

Стандартным решением в таком случае является преобразование алгоритма в целях использования только локальных данных или применение критических секций.

Использование только локальных данных для расчета в отдельном потоке является предпочтительным, так как при этом доступ к данным осуществляется монопольно из текущего потока и не нужно заботиться о синхронизации с другими потоками. В задаче с B-сплайнами размеры массивов left и right определяются на этапе выполнения программы и неизвестны на этапе компиляции. Поэтому они размещаются в динамической памяти, а не в стеке. На листинге 3 приведен соответствующий код.

Данный подход хотя и решает проблему, но влечет за собой дополнительные накладные расходы, связанные с необходимостью выделения динамической памяти с последующим ее освобождением при каждом вызове функции basis_funs.

```
void Bspline::basis_funs(int i, double u, double *b) const
{
   std::unique_ptr<double[]> left(new double[the_p + 1]);
   std::unique_ptr<double[]> right(new double[the_p + 1]);
   b[0] = 1.;
   for (int j = 1; j <= the_p; ++j)
   {
      left[j] = u - U[i + 1 - j];
      right[j] = U[i + j] - u;
…
```

Листинг 3. Использование для расчета только локальных данных

```
void Bspline::basis_funs(int i, double u, double *b) const
{
   b[0] = 1.;
   std::lock_guard<std::mutex> lg{mtx};
   for (int j = 1; j <= the_p; ++j)
   {
      left[j] = u - U[i + 1 - j];
      right[j] = U[i + j] - u;
…
```

Листинг 4. Применение критической секции

Применение критических секций позволяет обеспечить взаимно исключающий доступ к совместно используемым данным из разных потоков. В случае с B-сплайном придется поместить практически все тело функции basis_funs в критическую секцию, как показано на листинге 4.

Как видно, данный подход сводит на нет все преимущества параллельного выполнения функции basis_funs. Критическая секция, охватывающая весь код функции, фактически заставляет ее выполняться в последовательном режиме, попутно добавляя накладные расходы на работу объектов синхронизации.

Таким образом, стандартные решения в рассматриваемой задаче не обеспечивают удовлетворительного результата. Необходим механизм, позволяющий связать данные объекта с отдельным потоком так, что при обращении к функциям объекта из разных потоков использовалась бы отдельная копия данных, соответствующая текущему потоку. Далее предлагается одна из возможных реализаций такого механизма.

Использование локальной памяти потока — базовый алгоритм

Поставим задачу в общем виде. Имеется класс *O*, включающий объект данных класса *D*. Необходимо реализовать структуру класса *D* так, что для всех функций класса *O* будет использована отдельная копия объекта класса *D*, соответствующая потоку *T*, в котором выполняется функция. Схематически требуемая структура объектов при наличии нескольких потоков изображена на рисунке.

Схема на рисунке отображает тот факт, что отдельные копии данных создаются только в тех потоках, где они реально используются функциями конкретного объекта.

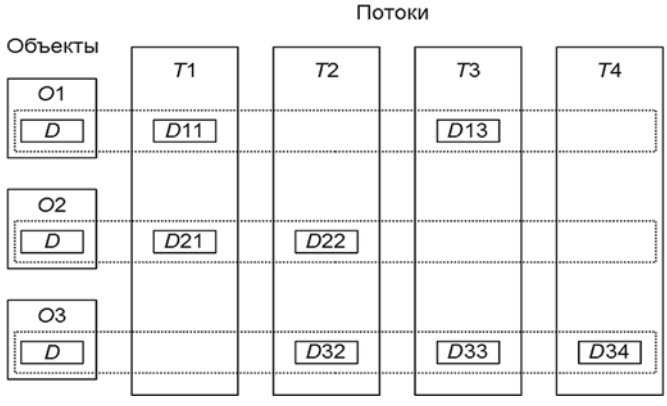
В указанной постановке задачи создать соответствующий механизм предоставления отдельных данных для каждого потока не составляет большого труда. Вначале возьмем для простоты в качестве данных целочисленную переменную. Потом обобщим разработанный алгоритм на выделение участка памяти произвольного размера для каждого потока. Возможная реализация в виде класса объектов с локальной копией целочисленной переменной для каждого потока приведена на листинге 5. На этом же листинге представлен пример использования класса.

В классе TLM ассоциативный массив vals содержит копии целочисленных переменных для каждого потока. При обращении к функциям SetVal и GetVal определяется текущий поток и сохраняется или считывается соответствующая этому потоку переменная. Функции SetVal и GetVal могут вызываться асинхронно из разных потоков, поэтому в них используется критическая секция для предотвращения повреждения структуры контейнера vals. Критическая секция действует только на время доступа к переменной, что существенно менее накладно по сравнению с кодом листинга 4, где блокировка охватывает весь расчетный алгоритм.

В приведенном на листинге 5 примере класс TLM включен в состав класса Obj. Объект tv обеспечивает функции класса Obj локальными по отношению к потокам данными. Так, в объекте obj создаются отдельные копии данных для основного потока и дочернего потока thr. В результате работы программа выдаст:

0
42

Приведенный алгоритм дает возможность связывать отдельные копии данных с конкретным потоком. Вместо целочисленных данных можно использовать указатели на размещаемые в динамической памяти объекты. Тогда при первом обращении к данным конкретного потока необходимо выделить память, а при уничтожении объекта, включающего



Отдельные копии данных для разных потоков

```
#include <iostream>
#include <mutex>
#include <thread>
#include <unordered_map>

using namespace std;

class TLM
{
 public:
   void SetVal(int val)
   {
   lock_guard<mutex> l(mtx);
   vals[this_thread::get_id()] = val;
   }

   int GetVal() const
   {
   lock_guard<mutex> l(mtx);
   auto p = vals.find(this_thread::get_id());
   return p == vals.end() ? 0 : p->second;
   }

 private:
   unordered_map<thread::id, int> vals;
   mutable mutex mtx;
};

class Obj
{
 public:
   void SetVal(int val) { tv.SetVal(val); }
   void PrintVal() { cout << tv.GetVal() << endl; }

 private:
   TLM tv;
};

Obj obj;

int main()
{
   obj.SetVal(42);
   auto thr = thread{ []{ obj.PrintVal();} };
   thr.join();
   obj.PrintVal();
}
```

Листинг 5. Соотнесение отдельной копии данных с каждым потоком

TLM, пройтись по всей цепочке контейнера vals и освободить выделенную для каждого потока память.

В рассмотренной реализации остался без внимания один вопрос. Что будет, если один из потоков закончит работу? Например, в схеме, представленной на рисунке, завершится поток *T*3. Тогда у объектов *O*1 и *O*3 останутся больше не нужные данные *D*13 и *D*33. Если объем данных невелик, а число потоков ограниченно (например, используется пул потоков), то это не является большой сложностью. Однако если размер данных значителен или потоки постоянно создаются и уничтожаются, то необходимо позаботиться об удалении из контейнера vals данных, соответствующих завершенным потокам. Таким образом, мы подходим к окончательной версии класса TLM, реализующего набор указателей на динамически выделенную память только для актуальных (т. е. незавершенных) потоков.

Использование локальной памяти с учетом актуальных потоков

Для того чтобы держать в составе объекта класса TLM только данные, соответствующие актуальным потокам, необходим некий механизм, который бы обеспечивал оповещение объектов TLM об окончании работы отдельных потоков. Для реализации такого механизма можно воспользоваться новым квалификатором памяти для глобальных и статических объектов, появившимся в стандарте языка C++ 11, — thread_local. Объект класса, объявленный с таким квалификатором, имеет свою отдельную копию по отношению к каждому потоку, где используется объект. При обращении к объекту берется та его копия, которая соответствует потоку, из которого происходит обращение. Таким образом, объект находится в локальной памяти потока.

Воспользовавшись этим свойством, можно создать объект в локальной памяти потока, который представляет собой коллекцию указателей на объекты TLM. Для каждого потока будет создана своя отдельная коллекция указателей. В каждой коллекции будут находиться только те объекты TLM, которые содержат данные для этого потока. Когда поток завершится, то вместе с ним будет удалена соответствующая локальная копия объекта. В этот момент можно пройти по всей цепочке объектов TLM, находящихся в коллекции завершаемого потока, и освободить данные, сопоставленные с этим потоком.

Для реализации такой стратегии создадим класс TLMset, обеспечивающий хранение и уничтожение коллекции указателей на объекты TLM. Единственный статический объект класса TLMset, объявленный с квалификатором thread_local, будет иметь свои копии, размещенные в локальной памяти каждого потока. На листинге 6 приведена первая часть файла tlm.cpp с кодом класса TLMset.

Класс TLMset обеспечивает внутренние детали реализации хранения указателей на используемые объекты TLM. Поэтому его описание не вынесено в заголовочный файл, а скрыто вместе с другим кодом, реализующим функции класса TLM.

Функция Add вызывается в классе TLM при добавлении данных для текущего потока. Функция Remove предназначена для использования в деструкторе класса TLM. Функции Add и Remove могут вызываться асинхронно из разных потоков, поэтому их тела помещены в критическую секцию. Если объект класса TLMset расположен в локальной памяти потока, то при завершении потока будет вызван деструктор

```
#include "tlm.h"
#include <unordered_set>

using namespace std;

class TLMset
{
 public:
  ~TLMset();

  void Add(TLM *ptlm)
  {
    lock_guard<mutex> l(mtx);
    ptlms.insert(ptlm);
  }

  void Remove(TLM *ptlm)
  {
    lock_guard<mutex> l(mtx);
    ptlms.erase(ptlm);
  }

 private:
  unordered_set<TLM*> ptlms;
  mutex mtx;
};

static thread_local TLMset tlmset;
```

Листинг 6. Реализация коллекции указателей на объекты TLM

~TLMset. В деструкторе происходит освобождение выделенной памяти для данных, сохраненных в объектах TLM и соответствующих завершаемому потоку. Код деструктора будет представлен после рассмотрения доработанной структуры класса TLM.

В соответствии с описанной стратегией необходимо доработать класс TLM так, чтобы при сохранении локальных данных потока указатель на объект TLM помещался в коллекцию TLMset текущего потока, а при удалении объекта TLM указатели на него исключались из коллекций TLMset всех потоков. На листинге 7 приведен заголовочный файл tlm.h с объявлением класса TLM.

Для соотнесения указателя на данные с конкретным потоком, как и прежде, используется ассоциативный массив. Только теперь вместе с указателем на данные в структуре TPtr сохраняется и указатель на коллекцию TLMset этого потока. Это позволит исключать указатели на удаляемые объекты TLM из коллекций TLMset всех потоков.

В состав класса TLM вошел указатель на задаваемую пользователем функцию fxnFreePtr, предназначенную для освобождения динамической памяти, соотнесенной с конкретным потоком. Динамическая память выделяется пользователем класса TLM для хранения в ней локальных данных потока. Указатель на выделенную память сохраняется в структуре TPtr как указатель на данные ptr для текущего потока.

Листинг 8 (вторая часть файла tlm.cpp) содержит определения функций класса TLM и деструктора класса TLMset.

Функция GetPtr выдает указатель на данные (т. е. динамически выделенную память), соотнесенные с вызвавшим ее потоком. Если данные для потока отсутствуют, то возвращается nullptr.

Функция SetPtr заносит указатель на данные в ассоциативный массив, соотнося их с текущим потоком. Вместе с указателем на данные сохраняется указатель на коллекцию tlmset для текущего потока. Статический объект tlmset объявлен с квалификатором памяти thread_local, поэтому для каждого потока будет создана его отдельная копия.

Служебная функция DeleteThisThreadPtr предназначена исключительно для использования в деструкторе класса TLMset для удаления данных из ассоциативного массива tptrs, соответствующих текущему (в нашем случае завершающемуся) потоку.

Выполнение деструкторов классов TLM и TLMset происходит в одной для всех объектов критической секции, обеспечиваемой глобальным мьютексом gMutex. Синхронизация потоков с использованием локальных блокировок посредством объектов mtx классов TLM и TLMset привело бы к взаимной бло-

```
#include <mutex>
#include <unordered_map>

class TLMset;

struct TPtr
{
  void *ptr;
  TLMset *ptlmset;
};

using FreePtrHandler = void (*)(void*);

class TLM
{
 public:
  TLM(FreePtrHandler fxnFreePtr_) : fxnFreePtr(fxnFreePtr_) {}
  ~TLM();
  void *GetPtr() const;
  void SetPtr(void *ptr);
  void DeleteThisThreadPtr();

 private:
  std::unordered_map<std::thread::id, TPtr> tptrs;
  FreePtrHandler fxnFreePtr;
  mutable std::mutex mtx;
};
```

Листинг 7. Объявление класса TLM

```
void *TLM::GetPtr() const
{
   lock_guard<mutex> l(mtx);
   auto p = tptrs.find(this_thread::get_id());
   return p == tptrs.end() ? nullptr : p->second.ptr;
}

void TLM::SetPtr(void *ptr)
{
   lock_guard<mutex> l(mtx);
   auto p = tptrs.find(this_thread::get_id());
   if (p == tptrs.end())
   {
      tlmset.Add(this);
      tptrs[this_thread::get_id()] = TPtr{ptr, &tlmset};
   }
   else
   {
      fxnFreePtr(p->second.ptr);
      p->second.ptr = ptr;
   }
}

void TLM::DeleteThisThreadPtr()
{
   lock_guard<mutex> l(mtx);
   auto p = tptrs.find(this_thread::get_id());
   if (p != tptrs.end())
   {
      fxnFreePtr(p->second.ptr);
      tptrs.erase(p);
   }
}

static mutex gMutex;

TLM::~TLM()
{
   lock_guard<mutex> l(gMutex);
   for (auto &p : tptrs)
   {
      fxnFreePtr(p.second.ptr);
      p.second.ptlmset->Remove(this);
   }
}

TLMset::~TLMset()
{
   lock_guard<mutex> l(gMutex);
   for(auto ptlm : ptlms) ptlm->DeleteThisThreadPtr();
}
```

Листинг 8. Реализация функций класса TLM

кировке при одновременном удалении объекта TLM и завершении работы потока. Деструктор ~TLM сначала бы зашел в свою критическую секцию, а потом, при вызове Remove, в критическую секцию объекта TLMset. В деструкторе ~TLMset порядок захода в критические секции противоположный: сначала секция объекта TLMset, а затем, при вызове DeleteThisThreadPtr, секция объекта TLM. Таким образом, создались бы классические условия для взаимной блокировки.

Насколько единая для всех объектов критическая секция будет снижать производительность, зависит от конкретного применения описанного инструмента. Предполагается, что объекты, использующие TLM, будут достаточно долгоживущими, например, будут находиться в составе некоторой рабочей базы данных. Тогда их удаление с соответствующим заходом в общую критическую секцию будет нечастым.

На листинге 9 представлен пример использования класса TLM.

В состав класса Obj включен объект buf класса TLM. Лямбда-функция, указанная при его иници-

```
#include "tlm.h"
#include <array>
#include <thread>

using namespace std;

class Obj
{
 public:
   Obj(int bufSize_) : bufSize{bufSize_} {}
   void Calc()
   {
      double *tlbuf = GetBuf();
      // Использование локальной к потоку памяти tlbuf.
   }

 private:
   TLM buf{[](void *p){ delete[] (double*)p; }};
   int bufSize;
   double *GetBuf()
   {
      double *p = (double*)buf.GetPtr();
      if (!p) buf.SetPtr(p = new double[bufSize]);
      return p;
   }
};

int main()
{
   array<thread, 5> fThreads;

   Obj obj{100};

   for (auto& th : fThreads)
   {
      th = thread {[&obj](){ obj.Calc(); }};
   }

   for (auto& th : fThreads)
      th.join();
}
```

Листинг 9. Пример использования класса TLM

```
#include <vector>
#include "tlm.h"

class Bspline
{
 public:
   Bspline::Bspline(const double *knots, int degree, int nmb)
      : the_p(degree), the_n(nmb - degree - 2), U(knots, knots + nmb)
   {}
   void basis_funs(int i, double u, double *b) const;

 private:
   std::vector<double> U;
   int the_n;
   int the_p;
   mutable TLM buf{[](void *p){ delete[] (double*)p; }};
   void GetLeftAndRight(double **pleft, double **pright) const
   {
      double *p = (double*)buf.GetPtr();
      if (!p) buf.SetPtr(p = new double[2 * (the_p + 1)]);
      *pleft = p;
      *pright = p + the_p + 1;
   }
};

void Bspline::basis_funs(int i, double u, double *b) const
{
   double *left, *right;
   GetLeftAndRight(&left, &right);
   b[0] = 1.;
   for (int j = 1; j <= the_p; ++j)
   {
      left[j] = u - U[i + 1 - j];
      right[j] = U[i + j] - u;
      double saved = 0.;
      for (int r = 0; r < j; ++r)
      {
         double temp = b[r] / (right[r + 1] + left[j - r]);
         b[r] = saved + right[r + 1] * temp;
         saved = left[j - r] * temp;
      }
      b[j] = saved;
   }
}
```

Листинг 10. Использование класса TLM для расчета значений B-сплайнов

ализации, выполняет освобождение динамической памяти, соотнесенной с отдельными потоками.

Функция GetBuf обеспечивает доступ к локальным данным потока. В представленном примере локальными данными является массив double размером bufSize. Для каждого потока выделяется отдельный массив в динамической памяти и соотносится с текущим потоком вызовом функции SetPtr. Если текущему потоку уже поставлен в соответствие выделенный массив в памяти, то GetPtr возвращает указатель на него.

B-сплайн с TLM-данными

Рассмотрев инструментальный механизм для работы с локальной памятью потока, вернемся к первоначальной задаче. На листинге 10 представлена реализация класса для расчета значений B-сплайнов, использующая для размещения вспомогательных массивов left и right локальную память потока.

Теперь функция basis_funs использует при выполнении только локальные по отношению к текущему потоку переменные. Блокировка в критических секциях происходит на очень короткое время для получения указателей на локальные данные. Код в основном теле функции не требует синхронизации с другими потоками и может выполняться в режиме параллельных вычислений.

Заключение

Представленный способ организации локальных к текущему потоку данных может быть использован в широком классе задач параллельного программирования. Разработанный класс TLM создает основу для дальнейших исследований в направлении уменьшения числа блокировок в многопоточном коде. В реализации класса TLM присутствуют критические секции. Наличие глобальной критической секции при завершении потока и удалении объекта TLM накладывает некоторые ограничения на сферу применения данного класса. С учетом этих ограничений наиболее благоприятной представляется следующая организация программной системы. В основном потоке происходит создание и удаление объектов, использующих TLM. Расчетные операции над объектами происходят в основном потоке, а также в других параллельно выполняющихся потоках. При этом для параллельных вычислений используется пул потоков. То есть для выполнения очередной расчетной задачи не создаются новые потоки, а берется один из существующих потоков пула. Пул потоков организуется, например, в библиотеке OpenMP и многопоточном программировании на основе задач в стандартной библиотеке C++ 11. При такой организации работы с объектами и потоками блокировка в упомянутой глобальной критической секции будет происходить крайне редко и не повлияет на общую производительность.

Список литературы

1. **Richter J. M., Nasarre C.** Windows via C/C++, Microsoft Press, 2007. 848 p.
2. **Stroustrup B.** C++ Programming Language, The 4th Edition, Addison-Wesley Professional, 2013. 1376 p.
3. **de Boor C.** A practical guide to splines, New York: Springer, 2001. 348 p.
4. **Piegl L., Tiller W.** The NURBS book, Berlin: Springer, 1997. 646 p.
5. **Gatlin K. S., Isensee P.** OpenMP and C++. Reap the benefits of multithreading without all the work // MSDN magazine, 2005, № 10. P. 17—28.

Using Thread Local Memory for Calculating B-Splines in Parallel Programming Tasks

P. M. Nikolaev, geom3d@gmail.com, Central Aerohydrodynamic Institute, Zhukovsky, 140181, Russian Federation

Corresponding author:

Nikolaev Prokopiy M., Head of Department, Central Aerohydrodynamic Institute, 140181, Zhukovsky, Russian Federation,

E-mail: geom3d@gmail.com

Received on November 29, 2018
Accepted on December 06, 2018

One of the main remaining problems in the multi-threaded programs development is the organization of asynchronous data access from different threads. The usage of locking objects is the standard method of synchronizing data access from different threads. Ensuring sequential access to shared data, locks reduces the degree of parallelism and decreases the overall performance of parallel algorithms. To reduce the number of locks, it is preferable to use a separate copy of the data for each running thread. The article discusses the method of dynamic allocation of memory associated with a specific thread. Thus, each executed thread has its own copy of the data and the need for synchronization disappears. The discussed method is implemented in the form of a C++ class that ensures the storage of a pointer to the memory local to the current thread and subsequent access to it. Pointers to the allocated memory are stored in an associative array. The access key is the identifier of the current thread. At the thread termination it is required to free the allocated memory associated with this thread. Based on the use of C++ thread_local qualifier, a special mechanism has been developed for freeing the memory associated with the thread when it is completed. An example of using the developed class in calculating the values of B-splines in a multithreaded environment is given. It is shown that the use of standard synchronization methods (stack variables, critical sections) when using auxiliary arrays in the B-spline calculation algorithm has significant overheads. The use of the developed method reduces the need for synchronization objects and improves the performance of calculating B-splines in multi-threaded programs.

Keywords: *parallel programming, multithreaded execution, asynchronous data access, local thread storage, B-spline*

For citation:

Nikolaev P. M. Using Thread Local Memory for Calculating B-Splines in Parallel Programming Tasks, *Programmnaya Ingeneria,* 2019, vol. 10, no. 4, pp. 178—185.

DOI: 10.17587/prin.10.178-185

References

1. **Richter J. M., Nasarre C.** *Windows via C/C++*, Microsoft Press, 2007, 848 p.
2. **Stroustrup B.** C++ *Programming Language*, The 4th Edition, Addison-Wesley Professional, 2013, 1376 p.
3. **de Boor C.** *A practical guide to splines*, New York, Springer, 2001, 348 p.
4. **Piegl L., Tiller W.** *The NURBS boo*k, Berlin, Springer, 1997, 646 p.
5. **Gatlin K. S., Isensee P.** OpenMP and C++. Reap the benefits of multithreading without all the work, *MSDN magazine*, 2005, no. 10, pp. 17—28.

УДК 004.4 DOI: 10.17587/prin.10.186-192

А. А. Скворцов, канд. техн. наук, доц. кафедры, e-mail: skvalexei@mail.ru,
Вятский государственный университет, г. Киров

Применение конечных автоматов при разработке пользовательского интерфейса встроенных систем

Рассмотрены вопросы, возникающие в процессе разработки пользовательского интерфейса встроенных систем. Показан способ применения конечных автоматов при разработке пользовательского интерфейса встроенных систем, при котором объем памяти, занимаемый программой, минимален.

Ключевые слова: *встроенная система, разработка, интерфейс, автоматное программирование, моделирование, конечный автомат, switch-технология, микроконтроллер*

Введение

Одной из технологий программирования, облегчающих разработку программного обеспечения встроенных систем, является автоматное программирование (или switch-технология). Согласно работе [1] программа микроконтроллера, разработанная в "автоматном" стиле, имеет ряд преимуществ по сравнению с программой, использующей функции операционной системы реального времени. Рассмотрим применение данной технологии программирования при разработке пользовательского интерфейса встроенных систем. Этот вариант применения автоматного программирования разработан автором для реализации собственных проектов и не является единственно возможным. Публикация может послужить примером для разработки аналогичных проектов пользовательского интерфейса.

В настоящее время при программировании пользовательского интерфейса встроенных систем управления, как правило, применяют запутанные вложенные или повторяющиеся циклические конструкции с различными флагами и переключателями. Такие конструкции усложняют чтение и особенно модификацию пользовательского интерфейса, а также приводят к непредусмотренным состояниям программы, которые довольно трудно найти и исправить при отладке. Благодаря явному выделению состояний (позиций) меню и условий перехода между ними автоматное программирование позволит навести порядок в программировании пользовательского интерфейса встроенных систем и построить надежные программы для вложенных многоуровневых пользовательских меню.

Работу в пользовательском меню можно смоделировать как одним, так и несколькими автоматами.

Целью исследования, результаты которого представлены в статье, является выбор оптимального способа применения конечных автоматов при разработке пользовательского интерфейса встроенных систем.

Основными критериями оптимизации программ являются быстродействие и объем занимаемой памяти. Программы пользовательского интерфейса по классификации Харела [2] относятся к интерактивным системам, которые некритичны к быстродействию, так как часто ожидают действий пользователя. Поэтому в настоящей работе критерием оптимизации будет объем занимаемой памяти.

Методика

Согласно работе [3] выделение управляющего состояния в автоматном программировании является сложной, неформальной задачей. Однако при разработке пользовательского меню эта задача значительно упрощается. При работе пользователя в меню настройки всегда существуют устойчивые позиции меню, отображаемые на экране или на другом устройстве вывода. Эти позиции и будут естественными управляющими состояниями программы, работающей с пользователем. Переходы между такими состояниями будут осуществляться в результате нажатий пользователем соответствующих кнопок. Рассмотрим пример проектирования пользовательского интерфейса программы управления грузоподъемным электромагнитом (ЭМ).

Допустим, для блока управления грузоподъемным ЭМ требуется разработать программу, выполняющую в нерабочем режиме следующие сервисные функции:

- весовые функции, а именно:
  - — просмотр весов погрузок в вагон;
  - — просмотр весов вагонов;
  - — копирование весов на SD-карту памяти и очистка весовой памяти;
- настройку сопротивления холодного магнита (от 1 до 8 Ом с шагом 0,1);
- настройку максимального напряжения на ЭМ (от 150 до 220 В);
- настройку времени сепарации, т. е. отделения листов металла при пониженном электрическом токе через ЭМ (от 5 до 20 с);

Обозначение	Сообщение на индикаторе	Описание
ГТ	Готов	Нерабочее состояние магнита
РБ	Работа	Рабочее состояние магнита
Н/С	Настройка/сервис	Главное меню
ВФ	Весовые функции	Пункт меню "Весовые функции"
ПГ	XXX) XXXX	Подпункт "Просмотр весов погрузок в вагон"
ВГ	XXX) XXXXX	Подпункт "Просмотр весов вагонов"
КП	Копир. и очистка	Подпункт "Копирование и очистка весовой памяти"
U_{max}	Макс.напр. XXXВ	Пункт меню "Максимальное напряжение"
$T_{сеп}$	Время сеп-и XXс	Пункт меню "Время сепарации"
Пар	* * * * *	Состояние ввода пароля
СХМ	Rхм = X.X Ом	Пункт меню "Сопротивление холодного магнита"
Тест	Тестовый режим/основной режим	Пункт меню "Основной/Тестовый режим"
ДТН	ДТ XXX ДН XXX	Пункт меню "Настройка датчиков"

- включение/выключение тестового режима;
- индикацию значений на датчиках тока и напряжения.

Настройка сопротивления холодного магнита, включение/выключение тестового режима и индикация значений на датчиках должны проводиться на заводе-изготовителе после ввода пароля, который состоит из пяти последовательных нажатий кнопок в определенной комбинации.

Просмотр весов осуществляется кнопками "→" (следующий вес) и "←" (предыдущий вес). Очистка SD-карты осуществляется кнопкой "←", а копирование на нее — кнопкой "→". Переход между подпунктами меню "Весовые функции" осуществляется кнопкой "•". Выход из подпунктов меню — кнопкой "*".

Каждый числовой параметр необходимо изменять с помощью кнопок "→" и "←", а сохранять с помощью кнопки "•". Выход из режима сервисных функций осуществляется кнопкой "E", переключение между настраиваемыми параметрами — кнопкой "*". Переход из нерабочего режима в рабочий и обратно осуществляется тумблером с двумя позициями "Вкл" ("1") и "Выкл" ("0").

Автоматное программирование начинается с выделения управляющих состояний. Настройка параметров возможна только при нерабочем состоянии магнита, называемом, например "Готов". Для отделения пользовательских настроек от параметров завода-изготовителя необходимо выделить верхний уровень меню, например "Настройка/сервис". При переходе в подменю "Настройка" нужно отобразить самый важный пользовательский параметр. Изменение параметра будет осуществляться кнопками "→" и "←", сохранение — кнопкой "•", переход к другому параметру — кнопкой "*".

При переходе в подменю "Сервис" нужно отобразить окно ввода пароля. Ввод пароля также можно оформить в виде автомата с пятью состояниями.

Далее параметры настраиваются аналогично пользовательским. Выход из меню и подменю в режим "Готов" осуществляется кнопкой "E".

Теперь можно выделить управляющие состояния. Перечень возможных управляющих состояний с их сокращенным обозначением и соответствующим сообщением на 16-разрядном алфавитно-цифровом индикаторе указан в таблице.

По условию задачи можно построить граф переходов в пользовательском меню. Работу пользователя в меню можно смоделировать как одним автоматом с 13 состояниями, так и несколькими автоматами. Рассмотрим эти два способа и выявим преимущества и недостатки каждого из них.

При моделировании первым способом граф переходов автомата выглядит так, как показано на рис. 1. Далее приведен код программы на языке Си, реализующий переходы этого графа.

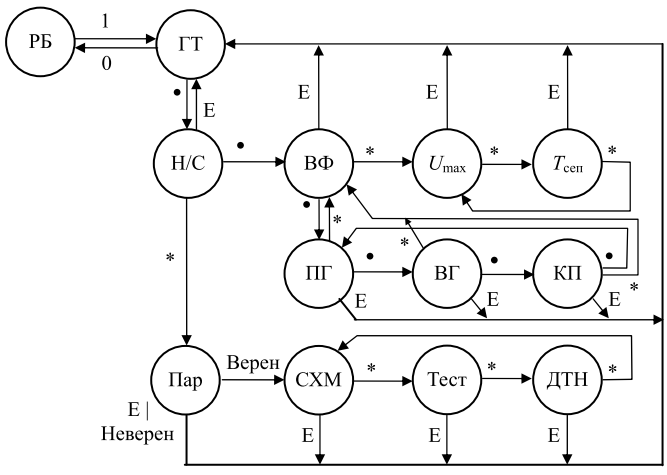


Рис. 1. Граф переходов автомата пользовательского меню при моделировании одним автоматом

```
#define NOKEY           0b00111110 //62 // Кнопка не нажата
#define LEFT            0b00111100 //60 // Нажата кнопка "←"
#define OK              0b00111010 //58 // Нажата кнопка "←"
#define RIGHT           0b00110110 //54 // Нажата кнопка "→"
#define STAR            0b00101110 //46 // Нажата кнопка "*"
#define ESC             0b00011110 //30 // Нажата кнопка "E"
#define SW              RB1// Тумблер
void main(void)
{
char s[17];             // Буфер вывода на LCD
char PSW[6];            // Массив символов пароля
char cnt=0;             // Счетчик символов пароля
char state=2;  // Состояние
char key=NOKEY;     // Код кнопки
...
while(1) {                      // Рабочий цикл
   key=ScanKbd();
   switch(state) {              // Код автомата
      case 1: // РБ — работа
        ...
        if(!SW) {lcd_putst("Готов"); state=2;}
        break;
      case 2: // ГТ — готов
        if(SW) {lcd_putst("Работа"); state=1;}
        else if(key == OK) {lcd_putst("Настройка/сервис"); state=3;}
        break;
      case 3: // Н/С — Настройка/сервис
        switch(key) {
           case OK: lcd_putst("Весовые функции"); state=4; break;
           case STAR: lcd_putst("Введите пароль"); state=10; break;
           case ESC: lcd_putst("Готов"); state=2; break;}
        break;
      case 4: // ВФ — весовые функции
        switch(key) {
           case OK: state=7; break;
           case STAR: state=5; break;
           case ESC: lcd_putst("Готов"); state=2; break;}
        break;
      case 5: // Umax — максимальное напряжение
        ...
        switch(key) {
           case STAR: state=6; break;
           case ESC: lcd_putst("Готов"); state=2; break;}
        break;
      case 6: // Tсеп — время сепарации
        ...
        switch(key) {
           case STAR: state=5; break;
           case ESC: lcd_putst("Готов"); state=2; break;}
        break;
      case 7: // ПГ — просмотр погрузок
        ...
        switch(key) {
           case OK: state=8; break;
           case STAR: state=4; break;
           case ESC: lcd_putst("Готов"); state = 2; break;}
        break;
      case 8: // ВГ — просмотр вагонов
        ...
        switch(key) {
```

```
            case OK: state=9; break;
            case STAR: state=4; break;
            case ESC: lcd_putst("Готов"); state=2; break;}
          break;
        case 9: // КП — копирование на SD-карту и очистка памяти
          ...
          switch(key) {
            case OK: state=7; break;
            case STAR: state=4; break;
            case ESC: lcd_putst("Готов"); state=2; break;}
          break;
        case 10: // Пар — ввод пароля
          ...
          if(key == ESC) {lcd_putst("Готов"); state=2;}
          else if(key!=NOKEY) {PSW[cnt]=key; cnt++;}
          if(cnt==5) {if(PSW=="<<<<<") state=11;
                      else {lcd_putst("Готов"); state=2;}}
          break;
        case 11: // СХМ — сопротивление холодного магнита
          ...
          switch(key) {
            case STAR: state=12; break;
            case ESC: lcd_putst("Готов"); state=2; break;}
          break;
        case 12: // Тест — переключение основной (0)/тестовый(1) режим
          ...
          switch(key) {
            case STAR: state=13; break;
            case ESC: lcd_putst("Готов"); state=2; break;}
          break;
        case 13: //ДТН — настройка датчиков
          ...
          switch(key) {
            case STAR: state=11; break;
            case ESC: lcd_putst("Готов"); state=2; break;}
      } // end switch(state)
      DelayMs(100); // Такт рабочего цикла
    } // end while(1)
    } // end main
```

В графе переходов опущены переходы при нажатии кнопок "→" и "←". Нажатие этих кнопок не приведет к переходу в другое состояние; оно связано с изменением значений параметров, что несущественно в данной работе.

Код автомата на языке Си, соответствующий графу переходов, представляет собой циклически выполняющийся блок switch-case, состоящий из обработчиков состояний и условий перехода между ними:

```
switch (state) {
    case 0: < действия в состоянии 0 >
            <условия перехода в другие состояния>
            break;
    case 1: < действия в состоянии 1 >
            <условия перехода в другие состояния>
            break;
    ...
    case n: < действия в состоянии n >
            <условия перехода в другие состояния>
}
```

Функция ScanKbd() выполняет сканирование кнопок (проверка на нажатие) и возвращает код кнопки. Функция lcd_putst() выводит указанное в скобках сообщение на жидкокристаллический индикатор. Функция DelayMs() задерживает выполнение программы на указанное в скобках число миллисекунд. Для корректного чтения нажатой кнопки необходима задержка не менее 100 мс, чтобы не прочитать ее дважды. Данная задержка является тактом работы автомата, так как определяет время между чтениями входных сигналов.

Теперь смоделируем работу в меню несколькими автоматами. В автомате, представленном на рис. 1, можно минимизировать число внутренних состояний. Согласно методу Ауфенкампа и Хона [4] минимизация автомата заключается в выделении и объединении k-эквивалентных состояний. В графе автомата на рис. 1 можно увидеть k-эквивалентные состояния, т. е. такие состояния, переход из которых под воздействием различных входных сигналов осуществляется в одну группу состояний. Сгруппируем

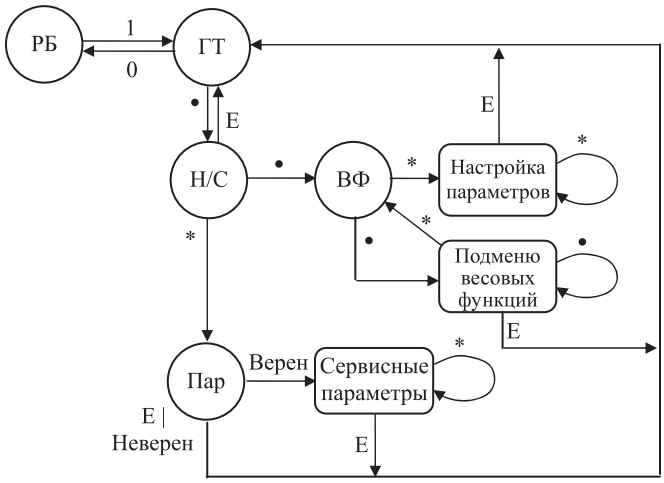


Рис. 2. Минимизированный граф переходов автомата пользовательского меню

в четыре группы: {РБ, ГТ, Н/С, ВФ, Пар}, {U_{max}, $T_{сеп}$}, {ПГ, ВГ, КП}, {СХМ, Тест, ДТН}. Тогда, например, переход из состояний U_{max}, $T_{сеп}$ второй группы по входному сигналу E осуществляется в состояние "ГТ" первой группы, а по входному сигналу * — в другие состояния второй группы. Следовательно, состояния второй группы являются k-эквивалентными и их можно объединить в одно состояние, а логику переходов между состояниями второй группы можно организовать в виде вложенного автомата или по счетчику. Аналогичные действия можно произвести с состояниями третьей и четвертой групп. На рис. 2 представлен минимизированный граф автомата с рис. 1, ниже представлен соответствующий автомату программный код. В главном автомате добавлены три состояния подменю: "Настройка параметров", "Подменю весовых функций", "Сервисные параметры", а восемь состояний (U_{max}, $T_{сеп}$, ПГ, ВГ, КП, СХМ, Тест, ДТН) перешли в автоматы, вложенные в указанные состояния подменю. Число состояний главного автомата сократилось с 13 до 8. Переключение между состояниями вложенных автоматов (состояниями подменю, т. е. состояниями групп 2, 3 и 4) реализовано по циклическим счетчикам state1, state2 и state3. Код пользовательского интерфейса значительно сократился вследствие сокращения числа состояний главного автомата, числа переходов между состояниями как главного меню, так и подменю. Повторение переходов в состояние ГТ сведено к минимуму.

```
while(1) {                              // Рабочий цикл
   key=ScanKbd();
   switch(state) {                      // Код автомата
      case 1: // РБ — работа
        ...
        if(!SW) {lcd_putst("Готов"); state=2;}
        break;
      case 2: // ГТ — готов
        if(SW) {lcd_putst("Работа"); state=1;}
        else if(key==OK) {lcd_putst("Настройка/сервис"); state=3;}
        break;
      case 3: // Н/С — Настройка/сервис
        switch(key) {
           case OK: lcd_putst("Весовые функции"); state=4; break;
           case STAR: lcd_putst("Введите пароль"); state=7; break;
           case ESC: lcd_putst("Готов"); state=2; break;}
        break;
      case 4: // ВФ — весовые функции
        switch(key) {
           case OK: state=6; break;
           case STAR: state=5; break;
           case ESC: lcd_putst("Готов"); state=2; break;}
        break;
      case 5: // Подменю "Настройка параметров"
        switch(state1) {
           case 0:... break; // Действия в состоянии Umax
           case 1:... break;} // Действия в состоянии Tсеп
        if(key==STAR) {state1=1-state1;}
        if(key==ESC) {lcd_putst("Готов"); state=2;}
        break;
      case 6: // Подменю "Весовые функции"
        switch(state2) {
           case 0: ... break; // Действия в состоянии ПГ
           case 1: ... break; // Действия в состоянии ВГ
           case 2: ... break;} // Действия в состоянии КП
        if(key==OK) {if(state2<2) state2++; else state2=0;}
```

```
        if(key==STAR) {lcd_putst("Весовые функции"); state=4;}
        if(key==ESC) {lcd_putst("Готов"); state=2;}
        break;
      case 7: // Пар — ввод пароля
        if(key==ESC) {lcd_putst("Готов"); state=2;}
        else if(key!=NOKEY) {PSW[cnt]=key; cnt++;}
        if(cnt==5) {if(PSW=="<<<<<") state=8; else {lcd_putst("Готов"); state=2;}}
        break;
      case 8: // Подменю "Сервисные параметры"
        switch(state3) {
           case 0: ... break; // Действия в состоянии СХМ
           case 1: ... break; // Действия в состоянии Тест
           case 2: ... break;} // Действия в состоянии ДТН
        if(key==STAR) {if(state3<2) state3++; else state3=0;}
        if(key==ESC) {lcd_putst("Готов"); state=2;}
      } // end switch(state)
      DelayMs(100); // Такт рабочего цикла
} // end while(1)
```

В результате минимизации автомата, составленного по техническому заданию, и выделения вложенных автоматов управляющий код пользовательского интерфейса сократился примерно в полтора раза (см. код исходного и минимизированного автоматов). После сборки кода автомата, представленного на рис. 2, объем занимаемой памяти программ оказался на 113 слов меньше, чем у кода автомата с рис. 1, что составляет 2,8 % памяти программ микроконтроллера PIC16F73, на котором проводилось исследование. В случае более сложного меню экономия памяти программ будет еще больше. Необходимо отметить, что для успешной минимизации исходного автомата, составленного по техническому заданию (как на рис. 1), условие выхода из подменю должно отличаться от условия перехода между пунктами подменю.

При моделировании действий пользователя одним автоматом код программы прост, он включает 13 однотипных обработчиков состояний, но при этом многократно повторяются условие перехода в главное меню, условие выхода из меню (case ESC: lcd_putst("Готов"); state=2; break;), а также условие перехода между пунктами подменю (case STAR: state=X; break;). В случае более сложного и многоуровневого меню число таких повторений может многократно возрасти.

При моделировании несколькими автоматами, несмотря на добавление состояний подменю ("Настройка параметров", "Весовые функции", "Сервисные параметры"), сокращается число состояний главного автомата, число переходов из подменю в главное меню, число выходов из меню, а также сокращаются переходы между пунктами подменю и, следовательно, значительно сокращается управляющий код пользовательского интерфейса. Уменьшается объем памяти программ, который во встроенных системах обычно очень ограничен.

Исходя из результатов исследования можно рекомендовать следующую последовательность проектирования пользовательского интерфейса встроенных систем с использованием конечных автоматов:

1) исходя из технического задания выделить состояния пользовательского меню и записать их в таблицу;

2) также исходя из технического задания определить группы состояний, соответствующих главному меню и различным подменю;

3) определить условия перехода между состояниями так, чтобы условия перехода между состояниями внутри групп совпадали, а условия выхода из групп (подменю) отличались от условий перехода внутри групп;

4) выделить дополнительные состояния, соответствующие группам подменю;

5) составить граф переходов главного автомата пользовательского меню;

6) при необходимости составить графы переходов автоматов подменю;

7) разработать код инициализации и код главного автомата, сопровождая их комментариями;

8) внутри состояний подменю разработать код вложенных автоматов подменю, используя, по возможности, циклические счетчики для переключения их состояний;

9) собрать программу в компиляторе, проставить точки останова в обработчиках состояний меню и подменю (напротив соответствующих "case") и проверить выполнение программы в соответствии с графом автомата и техническим заданием.

Код интерфейса, представленный в статье, можно использовать в качестве шаблона.

Заключение

Благодаря предварительной разработке графа конечного автомата применение автоматного подхода программирования позволило структурировать программный код пользовательского интерфейса, избегая запутанных циклических конструкций с различными флагами и переключателями. Построенный таким образом программный код пользовательского интерфейса легко читается и отлаживается, что повышает надежность разработки. Отладка такого программного кода проста: достаточно проставить точки останова в обработчиках состояний меню и подменю.

Исследование показало, что работу пользователя в меню эффективнее (с точки зрения объема занимаемой памяти) моделировать несколькими автоматами: главным, моделирующим работу в главном меню, и автоматами — обработчиками подменю, вложенными в состояние, соответствующее названию подменю.

С высокой долей уверенности можно предполагать, что представленный способ программирования пользо-

вательского интерфейса будет эффективен для многих аналогичных рассматриваемому примеру разработок.

Список литературы

1. **Татарчевский В.** Применение Switch-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 1 // Компоненты и технологии. 2006. № 11. URL: https://www.kit-e.ru/assets/files/pdf/2006_11_164.pdf

2. **Harel D., Pnueli A.** On the development of reactive systems // In "Logic and Models of Concurrent Systems". NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985. P. 477—498.

3. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование: Учебно-методическое пособие. СПб, 2007. 108 с.

4. **Aufenkamp D. D., Hohn F. E**. Analysis of Sequential Machines // IRE Trans. Electr. Cornput. 1957. Vol. 6. P. 276—283.

The Use of Finite State Machines in the Development Built-In Systems User Interface

A. A. Skvortsov, skvalexei@mail.ru, Vyatka State University, Kirov, 610000, Russian Federation

Corresponding author:

Skvortsov Aleksey A., Associate Professor**,** Vyatka State University, Kirov, 610000, Russian Federation, E-mail: skvalexei@mail.ru

Received on November 28, 2018
Accepted on January 21, 2019

The article discusses the simplification of the development of the user interface of embedded systems. The use of automata-based programming approach simplifies the development and improves the reliability of the user interface program of the built-in system. Simulating a user menu with multiple state machines reduces the memory footprint of the program code. Often at the present time in UI programming, embedded control systems apply the intricate nested or repeated cyclic design with various flags and switches. Such constructs make it difficult to read and especially modify the user interface, and lead to unexpected program states that are quite difficult to find and fix when debugging. Due to explicit state selection and transition between, automata-based programming will allow to restore order in UI programming and build reliable software for embedded multi-level custom menu. The aim of the work is to select the optimal method of finite state machines application in the development of the user interface of embedded systems. An example of the development of the user interface program for the control unit of a lifting electromagnet using automatic programming technology is given. We studied the simulation of the custom menu one state machine and multiple state machines. The use of automata-based programming made it possible to develop a simple, easy to read and easy to debug program code. Minimization of the graph automaton and the allocation of the sub-state machines have significantly reduced the amount of memory programs. Similarly, one can develop a program for the user interface of other control devices.

Keywords: *built-in system, development, interface, automata-based programming, switch-technology, modeling, state machine, microcontroller, UI programming, reliable software*

For citation:

Skvortsov A. A. The Use of Finite State Machines in the Development Built-In Systems User Interface, *Programmnaya Ingeneria*, 2019, vol. 10, no. 4, pp. 186—192.

DOI: 10.17587/prin.10.186-192

References

1. **Tatarchevskij V.** Primenenie Switch-tekhnologii pri razrabotke prikladnogo programmnogo obespecheniya dlya mikrokontrollerov. Chast 1 (The use of Switch-technology in the development of application software for microcontrollers. Part 1), *Komponenty i tekhnologii*, 2006, no. 11, available at: https://www.kit-e.ru/assets/files/pdf/2006_11_164.pdf (in Russian).

2. **Harel D., Pnueli A.** On the development of reactive systems, *In "Logic and Models of Concurrent Systems"*, NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems, Springer Verlag, 1985, pp. 477—498.

3. **Polikarpova N. I., Shalyto A. A.** *Avtomatnoe programmirovanie*: Uchebno-metodicheskoe posobie (Automata-based programming), Saint-Petersburg, 2007, 108 p. (in Russian).

4. **Aufenkamp D. D., Hohn F. E.** Analysis of Sequential Machines, *IRE Trans. Electr. Cornput.*, 1957, vol. 6, pp. 276—283.

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4
Технический редактор *Е. М. Патрушева.* Корректор *З. В. Наумова*

Сдано в набор 07.02.2019 г. Подписано в печать 21.03.2019 г. Формат 60×88 1/8. Заказ PI419
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
119071, г. Москва, Ленинский пр-т, д. 19, стр. 1. Сайт: **www.aov.ru**