УДК 681.3

А.Е. Дорошенко, К.А. Жереб, Е.А. Яценко

ФОРМАЛИЗОВАННОЕ ПРОЕКТИРОВАНИЕ ЭФФЕКТИВНЫХ МНОГОПОТОЧНЫХ ПРОГРАММ

Предложено совместное использование высокоуровневого инструментария алгеброалгоритмического проектирования, дополненного подсистемой переписывающих правил, а также низкоуровневых профилировщиков для автоматизации проектирования и повышения производительности последовательных и параллельных (многопоточных) программ с общей памятью.

Введение

Острая необходимость в повышении производительности программного обеспечения для решения трудоемких задач, с одной стороны, и новые возможности распараллеливания вычислений, предоставляемые многоядерными архитектурами современных микропроцессоров − с другой, побуждает к созданию специализированного инструментария для разработки параллельных программ для таких архитектур. Главным способом повышения производительности программ для таких платформ является распараллеливание программ с использованием многопоточности [1]. Проблема заключается не только в необходимости создания инструментов для проектирования новых программ, но также в обеспечении возможности реинженерии существующего программного обеспечения при его перенесении с устаревших на новые многоядерные платформы.

К известным инструментам, ориентированным на облегчение оптимизации параллельных программ с общей памятью, относятся, например, Intel Thread Profiler [2] и VTune [3] (последний предназначен для оптимизации последовательных программ). Однако они оба являются достаточно низкоуровневыми средствами, которые так и не получили широкого распространения, поскольку их использование требует достаточно высокой квалификации программиста и много ручной работы. Контролируя работу приложения, эти программы фиксируют возникающие проблемы, и после анализа полученных результатов тестирования программы, программист в итеративном режиме вручную должен модифицировать код программы, чтобы затем вновь анализировать ее выполнение с помощью профилировщика.

Разработка эффективных программ для многоядерных архитектур является масштабной научно-технической проблемой, успешное решение которой может быть обеспечено только путем специализации на предметные области и глубокого охвата этапов жизненного цикла разрабатываемых программ с применением средств автоматизации проектирования и программирования, от написания первоначальных спецификаций до генерации выполняемого кода. Основой для такой автоматизации является, прежде всего, высокоуровневая формализация конструирования многопоточных программ и автоматизация формальных трансформаций программ с целью оптимизации их производительности с применением профилировщиков только как элемента более высокой формализованной технологии разработки многопоточных программ. Под оптимизацией программ понимается достижение требуемых качественных характеристик программы по заданным критериям (память, быстродействие, загрузка оборудования и др.).

В Институте программных систем НАН Украины разработан интегрированный инструментарий проектирования и синтеза (далее «ИПС») программ [4], который базируется на представлении алгоритмов в системах алгоритмических алгебр (САА) В.М. Глушкова [5] и выполняет синтез последовательных и параллельных (многопоточных) программ на языках программирования Java и C++ по схемам алгоритмов. Для автоматизации

выполнения трансформаций алгоритмов (программ) предлагается интегрировать алгеброалгоритмический инструментарий «ИПС» с системой символьных вычислений (на основе парадигмы переписывания термов) TermWare [6–8], которая расширяет возможности создаваемого инструментального комплекса и позволит автоматизировать решение задач анализа и обеспечения надежности программного кода.

В данной работе описана методика совместного использования вышеназванных средств для формализованного проектирования и трансформации на уровне схем программ. На примере оптимизации программы с применением профилировщика Intel Thread Profiler для задачи нахождения простых чисел [9] продемонстрировано использование метаправил для автоматизации высокоуровневого проектирования.

1. Формализованное проектирование программ в интегрированном инструментарии

В разработанном интегрированном инструментарии проектирование алгоритмов осуществляется с помощью дерева конструирования в направлении сверху вниз путем детализации конструкций САА. По данному дереву автоматически строятся три формы представления алгоритма – естественно-лингвистическая, алгебраическая (регулярная схема) и графовая (граф-схема). В работе [4] детально рассмотрены алгебраическая и графовая формы, а также архитектура и интерфейс инструментария «ИПС». В данной работе используется естественно-лингвистическая форма (САА-схемы) алгоритмов, к преимуществам использования которой относятся, в частности, возможность описания алгоритмов в форме, удобной для человека, что облегчает достижение требуемого качества программ, а также ориентация на описание весьма сложных алгоритмических процессов посредством их поуровневого проектирования.

Основными объектами языка САА-схем (САА/1) [10], используемого в «ИПС», являются абстракции операторов (преобразователей данных) и условий (предикатов). Операторы и условия могут быть элементарными (базисными) или составными. *Базисный* оператор (условие) – это оператор (условие), который в САА-схеме считается первичной неделимой (атомарной) абстракцией. *Составные* операторы (условия) строятся из элементарных посредством операций последовательного и параллельного выполнения операторов:

- оператор1 ЗАТЕМ оператор2 – последовательное выполнение операторов;
- ЕСЛИ условие ТО оператор1 ИНАЧЕ оператор2 КОНЕЦ ЕСЛИ – оператор ветвления;
- ПОКА НЕ условие_окончания_цикла ЦИКЛ оператор КОНЕЦ ЦИКЛА – оператор цикла;
- ПАРАЛЛЕЛЬНО((*i*) = (1),..., (*n*))(оператор) – асинхронное выполнение *n* операторов (ветвей);
- ЖДАТЬ условие – синхронизатор, выполняющий задержку вычислений до момента, когда указанное условие (называемое условием синхронизации) станет истинным;
- КТ условие – контрольная точка, выполняющая установку условия (условия синхронизации) в истинное значение. Условия синхронизации, указанные в контрольных точках и синхронизаторах, связаны между собой;
- и других операций [5, 10].

Операторы и условия в САА-схемах помечаются смысловыми идентификаторами. Идентификатор некоторого оператора (соответственно условия) в САА-схеме – это обрамленная кавычками (соответственно апострофами) последовательность произвольной длины любых символов, за исключением кавычек (соответственно апострофов). Уровни в САА-схемах помечены левыми частями равенств, а структура каждого уровня конкретизирована (в терминах САА) правой частью соответствующего равенства. Левая часть равенства отделяется от правой цепочкой символов '='.

Пример 1. Далее приведена последовательная САА-схема алгоритма нахож-

дения простых чисел (данный алгоритм построен на основе метода, описанного в [9]), сконструированная в инструментарии «ИПС».

```
        СХЕМА НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ ====
        "Схема последовательного алгоритма нахождения общего количества простых чисел от 1 до 100,000 путем проверки каждого нечетного числа (number), является ли оно нацело делимым на меньшие нечетные факторы (factor)"
        КОНЕЦ КОММЕНТАРИЯ

"ОпределениеГлобальныхДанных"
==== "Определить константу (MAX_NUMBERS) = (100000)";
     "Определить массив (Primes) тип (long) размер (MAX_NUMBERS)";
     "Определить переменную (PrimeCount) тип (long)"

"ОсновнойСоставнойОператор"
==== "(PrimeCount) := (1)"
     ЗАТЕМ
     "(Primes[PrimeCount]) := (2)"
     ЗАТЕМ
     "Вывести сообщение (Определение простых чисел от 1 до ) и значение (MAX_NUMBERS)"
     ЗАТЕМ
     "Поиск"
     ЗАТЕМ
     "Вывести сообщение (Найдено простых чисел: ) и значение (PrimeCount)"

"Поиск"
==== ЛОКАЛЬНЫЕ_ПЕРЕМЕННЫЕ
     (
          "Определить переменную (number) тип (long)";
          "Определить переменную (start) тип (long)";
          "Определить переменную (end) тип (long)";
          "Определить переменную (stride) тип (long)";
          "Определить переменную (factor) тип (long)"
     )
     ЗАТЕМ
     "(start) := (1)"
     ЗАТЕМ
     "(end) := (MAX_NUMBERS)"
     ЗАТЕМ
     "(stride) := (2)"
     ЗАТЕМ
     ЕСЛИ '(start) = (1)'
     ТО "(start) := (start) + (stride)"
     КОНЕЦ ЕСЛИ
     ЗАТЕМ
     "(number) := (start)"
     ЗАТЕМ
     ПОКА НЕ '(number) >= (end)'
     ЦИКЛ
          "(factor) := (3)"
          ЗАТЕМ
          ПОКА НЕ 'Остаток от деления (number) на (factor) = (0)'
          ЦИКЛ
               "(factor) := (factor) + (2)"
          КОНЕЦ ЦИКЛА
          ЗАТЕМ
          ЕСЛИ '(factor) = (number)'
          ТО "(Primes[PrimeCount]) := (number)"
          ЗАТЕМ
          "(PrimeCount) := (PrimeCount) + (1)"
          КОНЕЦ ЕСЛИ
          ЗАТЕМ
          "(number) := (number) + (stride)"
     КОНЕЦ ЦИКЛА

КОНЕЦ СХЕМЫ НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ
```

Отметим, что по рассмотренной схеме в инструментарии «ИПС» была сгенерирована программа на языке C++. Процесс распараллеливания данного алгоритма приведен в подразделе 2.1 в качестве примера оптимизации алгоритма для выполнения на мультипроцессорной архитектуре.

2. Оптимизация программ на основе использования инструментария «ИПС», TermWare и Intel Thread Profiler

Инструментарий «ИПС» может быть дополнен средствами преобразования схем последовательных и параллельных алгоритмов, направленных на их улучшение. Преобразование программ базируется на метаправилах трансформации и переориентации [10], используемых в алгебрах алгоритмов, а также применении системы переписывания термов TermWare [7].

Метаправило трансформации состоит в применении к схемам равенств вида $t_1 = t_2(x_1, x_2, ..., x_s)$, где t_1 и t_2 – термы в рассматриваемой алгебре алгоритмов, зависящие от переменных $x_1, x_2, ..., x_s$. К равенствам относятся тождества, квазитождества и соотношения [10]. Тождества представляют собой равенства, справедливые при любых значениях переменных $x_1, x_2, ..., x_s$. Тождества в алгебре применяются к формулам в направлении слева направо или наоборот, справа налево. Квазитождества – равенства, которые выполняются лишь при некоторых интерпретациях, входящих в них переменных. Соотношения характеризуют особенности выбранной предметной области [10].

Переориентация алгоритма состоит в последовательном применении к алгоритму метаправил свертки и развертки, и базируется на использовании систем равенств вида $I = \{v_1 = T_1, v_2 = T_2, \ldots, v_r = T_r\}$, где v_i – операторные или предикатные переменные, T_i – термы в алгебре алгоритмов. Свертка алгоритма представляет собой замену вхождений в схему непересекающихся правых частей равенств T_i на соответствующие левые части v_i и направлена на абстрагирование схемы. Развертка алгоритма состоит в замене переменных v_i схемы на соответствующие термы T_i и ориентирована на конкретизацию схемы. Частным случаем переориентации является переинтерпретация схемы – замена ее базисных элементов (операторов и предикатов).

В инструментарии «ИПС» рассмотренные равенства могут применяться в автоматизированном режиме, для чего разрабатывается библиотека равенств, механизм их выбора и применения.

Далее приведен пример оптимизирующего преобразования – трансформация последовательного алгоритма в параллельный (на примере задачи нахождения простых чисел [9]), а также рассмотрена оптимизация полученного параллельного алгоритма (и соответствующей ему программы) с использованием алгеброалгоритмического аппарата и системы Intel Thread Profiler и системы TermWare для выполнения трансформаций схем.

2.1. Оптимизация 1.

Распараллеливание последовательного алгоритма. Рассмотрим трансформацию последовательного алгоритма, осуществляющего циклическую обработку одномерной последовательности данных D длиной n, в соответствующий параллельный алгоритм с m параллельными ветвями ($m \geq 1$). Для параллельной обработки последовательность D разделяется на m приблизительно равных частей (блоков). Каждая ветвь выполняет обработку блока длиной $b = n / m$. Указанная трансформация может быть представлена в САА-М в виде следующего равенства:

$$Serial(D) = Parallel(D), \quad (1)$$

где

Serial(*D*) ::= "(*i*) := (*start*)" (1.1)
ЗАТЕМ
ПОКА НЕ '(*i*) >= (*end*)'
ЦИКЛ
ОБРАБОТКА_ДАННЫХ(*D*, *i*)
КОНЕЦ ЦИКЛА;

Parallel(*D*) ::= ПАРАЛЛЕЛЬНО((*j*) = (0),..., (*m*–1)) (1.2)
(
ВЕТВЬ(*j*)
)
ЗАТЕМ
ЖДАТЬ 'Обработка во всех (*m*) ветвях закончена';

ВЕТВЬ(*j*) ::= “(*start*) := (*j* * *b* + 1)”
ЗАТЕМ
“(*end*) := (*j* * *b* + *b*)”
ЗАТЕМ
“(*k*) := (*start*)”
ЗАТЕМ
ПОКА НЕ '(*k*) >= (*end*)'
ЦИКЛ
ОБРАБОТКА_ДАННЫХ(*D*, *k*)
КОНЕЦ ЦИКЛА
ЗАТЕМ
КТ 'Обработка в ветви (*j*) закончена';

Здесь *Searial*(*D*), *Parallel*(*D*) – операторы, задающие соответственно стратегии последовательной и параллельной обработки; *i*, *j*, *k* – индексы; *start*, *end* – начальные и конечные значения индексов *i*, *k*; ОБРАБОТКА_ДАННЫХ(*D*, *i*) – оператор, выполняющий обработку данных *D* в соответствии со значением индекса *i*, а также изменяющий значение индекса *i* в соответствии с алгоритмом (приращение или уменьшение значения индекса); ВЕТВЬ(*j*) – оператор, реализующий *j*-ю ветвь вычислений ($j = 0, \ldots, m - 1$). Для синхронизации ветвей в схеме использованы контрольная точка и синхронизатор (см. раздел 1).

Для параллельных программ над общей памятью важной является также синхронизация параллельных ветвей, касающаяся защиты совместно используемых ресурсов. В САА-М для синхронизации фрагмента алгоритма, осуществляющего доступ к разделяемым ресурсам, данный фрагмент необходимо включить в блок критической секции, т. е. применить к схеме следующее равенство:

$$U = \text{КритическаяСекция}\ (cs)\ (\ U\ ), \quad (2)$$

где *U* – оператор, выполняющийся в параллельной ветви и осуществляющий доступ к разделяемым ресурсам; *cs* – идентификатор критической секции.

В некоторых языках программирования требуется также предварительная инициализация критической секции. В этом случае применяется следующее равенство:

V = ИнициализироватьКритическуюСекцию(*cs*) ЗАТЕМ *V*, (3)

где *V* – оператор, перед которым необходимо выполнить данную инициализацию.

Пример 2. На основе равенств (1) – (3) был осуществлен переход от схемы последовательного алгоритма определения простых чисел (см. раздел 1, пример 1) к соответствующей параллельной схеме. В схему были внесены также и другие изменения: 1) схема дополнена глобальными константами и переменными NUM_THREADS, BLOCKSIZE, LONG_CACHE, factor, ThreadNums, ThreadHandles, Primes_CS; 2) в функцию Поиск введен параметр ThreadNum; 3) для вызова функции Поиск(ThreadNum) из отдельной ветви используется промежуточная функция НайтиПростыеЧисла(Arg); 4) введена локальная переменная space в функцию Поиск(ThreadNum); 5) вместо переменной factor введена замена текста factor на space[2 * LONG_CACHE] в алгоритме; 6) необходимым образом изменены значения переменных start и end в функции Поиск(ThreadNum). Переменные LONG_CACHE, space и промежуточная функция НайтиПростыеЧисла(Arg) вводятся в параллельной программе для исключения при выполнении программы проблемы “64k aliasing” (более подробно см. [2]).

Полученная в результате трансформации параллельная схема имеет следующий вид:

```
СХЕМА НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ/П ====
        "Схема параллельного алгоритма нахождения простых чисел от 1 до 100,000"
КОНЕЦ КОММЕНТАРИЯ

"ОпределениеГлобальныхДанных"
==== "Определить константу (NUM_THREADS) = (4)";
      "Определить константу (MAX_NUMBERS) = (100000)";
      "Определить константу (BLOCKSIZE) = ((MAX_NUMBERS / NUM_THREADS))";
      "Определить константу (LONG_CACHE) = (64 / sizeof(long))";
      "Определить замену (factor) на (space[2 * LONG_CACHE])";
      "Определить массив (ThreadNums) тип (long) размер (NUM_THREADS)";
      "Определить массив (ThreadHandles) тип (HANDLE) размер (NUM_THREADS)";
      "Определить массив (Primes) тип (long) размер (MAX_NUMBERS)";
      "Определить переменную (PrimeCount) тип (long)";
"Определить критическ"ОсновнойСоставнойОператор"

==== "ИнициализироватьКритическуюСекцию(Primes_CS)"

      ЗАТЕМ
      "(PrimeCount) := (1)"
      ЗАТЕМ
      "(Primes[PrimeCount]) := (2)"
      ЗАТЕМ
      "Вывести сообщение (Определение простых чисел от 1 до ) и значение (MAX_NUMBERS)"
      ЗАТЕМ
      ПАРАЛЛЕЛЬНО((thread) = (0),..., (NUM_THREADS - 1))
      (
          "НайтиПростыеЧисла(thread)"
      )
      ЗАТЕМ
      ЖДАТЬ 'Обработка во всех (NUM_THREADS) ветвях закончена'
      ЗАТЕМ
      "Вывести сообщение (Найдено простых чисел: ) и значение (PrimeCount)"

"НайтиПростыеЧисла(Arg)"
==== "Подготовить данные (ThreadNum) для вызова функции"
      ЗАТЕМ
      "Поиск(ThreadNum)"
```

```
"Поиск(ThreadNum)"
==== ЛОКАЛЬНЫЕ_ПЕРЕМЕННЫЕ
      (
            "Определить переменную (number) тип (long)";
            "Определить переменную (start) тип (long)";
            "Определить переменную (end) тип (long)";
            "Определить переменную (stride) тип (long)";
            "Определить массив (space) тип (long) размер (4 * LONG_CACHE)"
      )

      ЗАТЕМ
      "(start) := (ThreadNum * BLOCKSIZE + 1)"
      ЗАТЕМ
      "(end) := (ThreadNum * BLOCKSIZE + BLOCKSIZE)"
      ЗАТЕМ
      "(stride) := (2)"
      ЗАТЕМ
      ЕСЛИ '(start) = (1)'
      ТО "(start) := (start) + (stride)"
      КОНЕЦ ЕСЛИ
      ЗАТЕМ
      "(number) := (start)"
      ЗАТЕМ
      ПОКА НЕ '(number) >= (end)'
      ЦИКЛ
            "(factor) := (3)"
            ЗАТЕМ
            ПОКА НЕ 'Остаток от деления (number) на (factor) = (0)'
            ЦИКЛ
                  "(factor) := (factor) + (2)"
            КОНЕЦ ЦИКЛА
            ЗАТЕМ
            ЕСЛИ '(factor) = (number)'
            ТО КритическаяСекция (cs)
            (
                  "(Primes[PrimeCount]) := (number)"
                  ЗАТЕМ
                  "(PrimeCount) := (PrimeCount) + (1)"
            )
            КОНЕЦ ЕСЛИ
            ЗАТЕМ
            "(number) := (number) + (stride)"
      КОНЕЦ ЦИКЛА
      ЗАТЕМ
      КТ 'Обработка в ветви (ThreadNum) закончена'

КОНЕЦ СХЕМЫ НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ/П
```

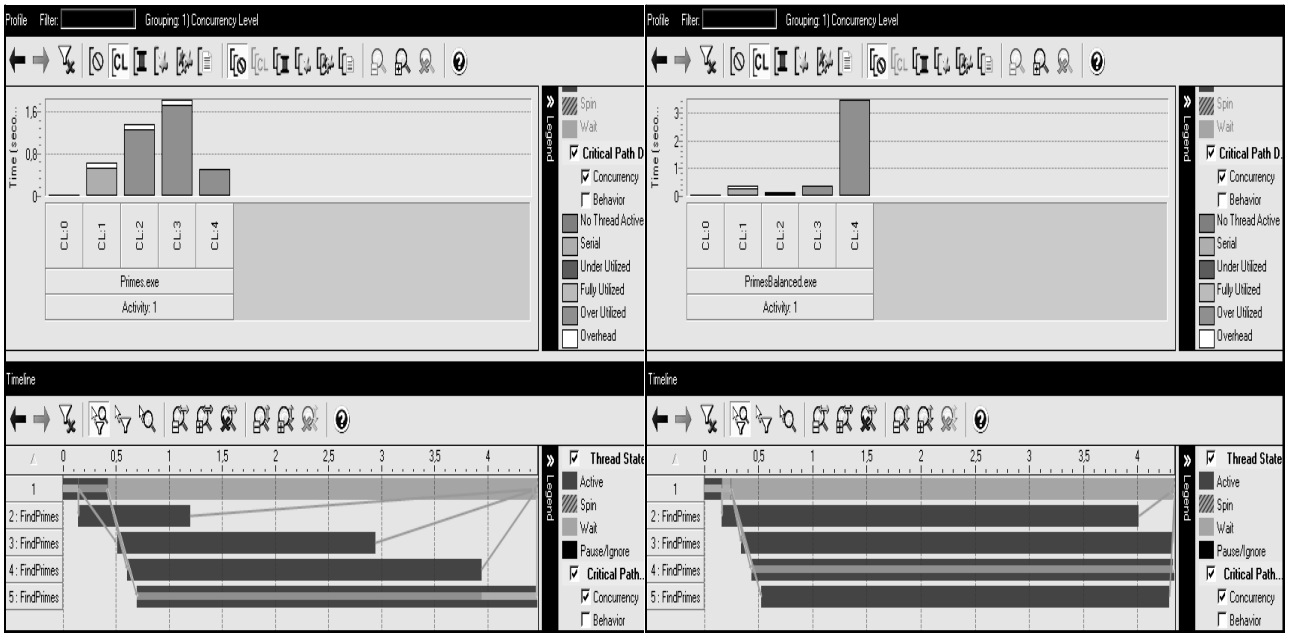
По приведенной параллельной схеме алгоритма в инструментарии «ИПС» была сгенерирована многопоточная программа на языке C++.

2.2. Оптимизация параллельной программы. Для оптимизации параллельных программ инструментарий «ИПС» может совместно использоваться с системой Intel Thread Profiler. Thread Profiler, контролируя работу многопоточного приложения, фиксирует все возникающие проблемы, включая рассинхронизацию потоков и накладные расходы на переключение между потоками. Полученные данные отображаются в графическом виде, который позволяет быстро идентифицировать фрагменты исходных текстов, нуждающиеся в доработке. Далее в «ИПС» и TermWare осуществляется автоматизированное преобразование алгоритма, направленное на улучшение программы. После чего синтезированная программа вновь анализируется в Thread Profiler для проверки того, что проблема решена.

Оптимизация 2.1. Сбалансированное распределение работы по потокам. В выше-рассмотренной схеме параллельного алгоритма большие числа требуют больше времени для определения того, являются ли они простыми [9], поэтому последовательное разбиение области вычислений на части для обработки каждой из параллельных ветвей приводит к неэффективному использованию ресурсов процессора. Выполнение данной программы было проанализировано с помощью Thread Profiler на ПК с ОС MS Windows XP, процессор Intel Pentium 4 2.34 ГГц без поддержки технологии Hyper-Threading. На рис. 1, а показано окно Thread Profiler с полученными Profile диаграммой (верхняя половина окна) и временной диаграммой (нижняя половина окна). На Profile диаграмме показаны итоговые данные о времени, потраченном потоками программы на критическом пути. При этом результаты сгруппированы по категории “уровень параллелизма” (concurrency level), которая обозначает количество одновременно выполняющихся активных потоков (от 0 до 4). Временная диаграмма иллюстрирует поведение программы с течением времени и показывает вклад каждого потока в программу в целом. В Thread Profiler отображены различными оттенками монохромного изображения последовательные участки кода (Serial); превышение в данном приложении количества потоков над количеством процессоров в системе (Over Utilized); время, требуемое для передачи управления от одного потока к другому (Overhead). На временной диаграмме отдельными оттенками показаны активные потоки (Active), фазы ожидания потоков (Wait), стрелками показаны разделения (Fork) и слияния (Join) потоков.

Как видно из приведенной на рис. 1, а временной диаграммы, с увеличением номера потока увеличивается время его выполнения, т. е. загрузка потоков в данной программе является несбалансированной. С целью улучшения алгоритма массив чисел разделяется таким образом, чтобы блоки, обрабатываемые потоками, были вложенными (перемежающимися). При этом каждая из ветвей будет работать как на малых, так и на больших числах.

Для указанной оптимизации алгоритма применим к нему метаправило переинтерпретации, использующее системы равенств I_1 и I_2:



а б

Рис. 1. Результаты анализа в Thread Profiler многопоточной программы определения простых чисел: а – до оптимизации; б – после оптимизации

I_1 = { v_1 = “(start) :=
=(ThreadNum * BLOCKSIZE + 1)”;
v_2 = “(end) :=
= (ThreadNum * BLOCKSIZE +
+ BLOCKSIZE)”;
v_3 = “(stride) := (2)”; };

I_2 = { v_1 = “(start) := (2 * ThreadNum + 1)”;
v_2 = “(end) := (MAX_NUMBERS)”;
v_3 = “(stride) := (2* NUM_THREADS)”; },

где v_1, v_2, v_3 – операторные переменные.

Вначале к алгоритму применяются равенства системы I_1 в направлении справа налево (базисные операторы заменяются на переменные v_1, v_2, v_3). Затем вместо указанных переменных подставляются уже другие базисные операторы путем применения равенств системы I_2 слева направо. В результате получаем оптимизированный алгоритм, в котором работа по нахождению простых чисел равномерно распределена между потоками.

По данному алгоритму был выполнен синтез программы в «ИПС», результаты анализа которой показаны на рис. 1, б. На Profile диаграмме видно, что большее время занимает одновременное выполнение всех 4 потоков, а временная диаграмма показывает улучшенный баланс загрузки потоков по сравнению с предыдущим вариантом программы. Таким образом, увеличилась совокупная часть времени параллельного выполнения программы по отношению к последовательной части. Тем не менее, данная программа все еще требует улучшения – на Profile диаграмме изображены издержки при передаче управления от одного потока к другому, которые становятся видны при увеличении масштаба диаграммы.

Оптимизация 2.2. Исключение накладных расходов при передаче управления от одной параллельной ветви к другой. Рассматриваемая в данном пункте трансформация ориентирована исключение временных издержек, связанных со входом и выходом из критической секции в многопоточной программе. Трансформация связана с наличием в Win32 API специализированного средства синхронизации – группы особых функций, названия которых начинаются с префикса Interlocked. Суть их в том, что каждая из них позволяет выполнить пару простых операций (одну из арифметических (логических) операций и операцию проверки полученного значения), таким образом, что они выполняются атомарно и их выполнение не может быть прервано другим потоком. Если критическая секция в программе содержит операции, которые могут быть заменены соответствующими Interlocked-функциями, то это позволит улучшить выполнение приложения [9]. Указанное преобразование можно представить следующим равенством в алгебре алгоритмов:

КритическаяСекция (*cs*)
(АтомОп(x_1, x_2, …, x_n)) =
= БлокАтомОп(x_1, x_2, …, x_n),

где АтомОп(x_1, x_2, …, x_n) – арифметическая или логическая операция; БлокАтомОп(x_1, x_2, …, x_n) – блокирующий аналог указанной операции, представляемый в языке программирования соответствующей Interlocked-функцией.

Пример 3. Необходимо исключить критическую секцию в следующем фрагменте алгоритма определения простых чисел (см. пример 2):

```
КритическаяСекция (cs)
(
   "(Primes[PrimeCount]) := (number)"
   ЗАТЕМ
   "(PrimeCount) := (PrimeCount) + (1)"
)
```

Шаг 1. Приведение композиции двух операторов к одному оператору. Заменяем второй базисный оператор на “ИНК(PrimeCount)” (применяем метаправило переинтерпретации). Затем подставляем базисный оператор “ИНК(PrimeCount)” вместо PrimeCount в Primes[PrimeCount] (при этом предполагается, что значение выражения ИНК(PrimeCount) является значением PrimeCount до того, как применено приращение):

```
КритическаяСекция (cs)
(
  "(Primes[ИНК(PrimeCount)]) := (number)"
)
```

Шаг 2. Заменяем операцию приращения ИНК на соответствующую блокирующую операцию и исключаем критическую секцию:

```
"(Primes[БЛОК_ИНК(PrimeCount)]) :=
                = (number)"
```

По оптимизированному таким образом алгоритму в «ИПС» была сгенерирована программа, выполнение которой было проанализировано в Thread Profiler. По сравнению с предыдущими результатами (см. рис. 1, б), накладные расходы на передачу управления между потоками исключены в столбцах Profile диаграммы, отмеченных CL:2 и CL:3.

2.3. Использование системы TermWare для трансформации алгоритмов. Для применения равенств к схемам алгоритмов «ИПС» может применяться совместно с системой символьных вычислений TermWare [6–8]. Система TermWare – открытая структура перезаписи термов, которая состоит из двух главных частей:

1) библиотеки Java, содержащей основные структуры данных и алгоритмы для перезаписи термов, правила перезаписи термов, унификации, стратегии перезаписи и пр.;

2) структуры Java, содержащей средства добавления синтаксических анализаторов, языков программирования и правил перезаписи с действиями, которые могут быть введены в прикладные программы Java и могут быть расширены через структуру Java.

Главное отличие TermWare от обычных систем перезаписи термов состоит в том, что эта система перезаписи термов – не „замкнутая” формальная система в том смысле, что она не предназначена для обеспечения полной среды программирования, и не объединена в пакет как интерпретатор или транслятор, а представляется как библиотека, встраиваемая в программное приложение.

Для разработчика система термов выглядит как совокупность экземпляров класса ITermSystem, с возможностью управления набором правил и редуцирования термов [8]. База фактов также может содержать императивные элементы, описанные как Java классы. Таким образом, имеется возможность использовать декларативную модель программирования в программных комплексах, основанных на Java-платформе в тех случаях, когда это необходимо. Разработчик может использовать термальную систему как программный агент, встраиваемый в общую инфраструктуру программной системы.

Сами наборы правил описаны на языке TermWare и хранятся в отдельных файлах. Основой языка являются термы, т.е. выражения вида $f(x_1, ..., x_n)$. В качестве атомарных термов используются переменные (которые записываются в виде $identifier), а также константы определенных типов данных (числовых, логических, строковых и атомарного – неизменяемые строки). Для упрощения записи и восприятия используются сокращения для многих термов, например, $x+y$ – для $plus(x, y)$; $x\ ?\ y:\ z$ – для $ifelse(x, y, z)$ и др. Весь набор правил является термом, записанным с использованием этих сокращений. Набор правил содержит сами правила, а также дополнительную информацию (название системы правил, используемая база фактов, применяемая стратегия). Типичное правило в TermWare записывается следующим образом:

$$source\ [condition] \rightarrow destination\ [action].$$

Здесь используются четыре терма: source – входной образец; destination – выходной образец; condition – условие, определяющее применимость правила; action – действие, выполняемое при срабатывании правила. Выполняемые действия и проверяемые условия в основном являются вызовами методов в БД фактов. Таким образом, осуществляется связь между правилами на языке TermWare и кодом на Java. Кроме того, возможно написание собственной стратегии, определяющей порядок применения правил.

В результате интеграции системы TermWare и разработанного инструментария проектирования и синтеза программ, в TermWare может передаваться дерево алгоритма, подлежащего трансформации и требуемые для этого равенства. После выполненного в TermWare преобразования алгоритма, его дерево будет передано обратно в «ИПС». Далее по оптимизированному алгоритму может быть произведена генерация соответствующей программы на языке программирования.

Пример 4 (трансформация последовательного алгоритма в параллельный с использованием TermWare). Вышеприведенная трансформация последовательного алгоритма в параллельный может быть автоматизирована с использованием TermWare. Исходная схема последовательного алгоритма записывается в виде терма:

```
Root(
 Globals(
  THEN(
   Constant(MAX_NUMBERS,100000),
   THEN(
    Array(Primes,long,MAX_NUMBERS), Variable(PrimeCount,long)))),
 Main(
  THEN(
   Assignment(PrimeCount,1),
   THEN(
    Assignment( ArrayElement(Primes,PrimeCount),2),
    THEN(Print,
     THEN(
      CALL(Search),
      THEN(Print,
       END(Main)))))))),
 Search(
  THEN(
   LocalVariables(
    THEN(
     Variable(number,long),
     THEN(
      Variable(start,long),
      THEN(
       Variable(end,long),
       THEN(
        Variable(stride,long),
        Variable(factor,long)))))),
   THEN(
    Assignment(start,1),
    THEN(
     Assignment(end,MAX_NUMBERS),
     THEN(
      Assignment(stride,2),
      THEN(
       IF(
        eq(start,1),
        Assignment(start,  plus(start,stride))),
       THEN(
        Assignment(number,start),
        THEN(
         UNTIL(
          greater_eq(number,end),
          THEN(
           Assignment(factor,3),
           THEN(
            UNTIL(
             eq(
              mod(number,factor),0),
             Assignment(factor,    plus(factor,2))),
            THEN(
             IF(
              eq(factor,number),
              THEN(
               Assignment(   ArrayElement(Primes,PrimeCount),number),
               Assignment(PrimeCount,  plus(PrimeCount,1)))),
             Assignment(number,  plus(number,stride)))))),
         END(Search)))))))))))
```

Этот терм соответствует схеме последовательного алгоритма из примера 1. (Поскольку язык TermWare не позволяет использовать русскоязычные обозначения термов, названия элементов схемы заменены на соответствующие англоязычные). Отдельные уровни схемы ("ОпределениеГлобальныхДанных", "ОсновнойСоставнойОператор", "Поиск") представлены в виде подтермов терма Root (Globals, Main, Search). Для записи арифметических действий использованы стандартные термы системы TermWare, что позволяет вводить базисные операторы и условия в естественном виде. Например, выражение ThreadNum * BLOCKSIZE + 1 автоматически трансформируется в терм plus(multiply(ThreadNum,BLOCKSIZE),1).

В качестве примера записи правил рассмотрим первый этап трансформации (распараллеливание циклов, без устранения проблемы “64k aliasing”). При этом применяется следующая совокупность правил:

1. Globals(THEN(Constant(MAX_NUMBERS,100000),$0))->
   Globals(THEN(THEN(Constant(MAX_NUMBERS,100000),$0),
   THEN(Constant(NUM_THREADS,4),
   THEN(Constant(BLOCKSIZE,divide(MAX_NUMBERS,NUM_THREADS)),
   CriticalSection(Primes_CS)))))
2. Main(THEN(Assignment(PrimeCount,1),$0)) ->
   Main(THEN(InitCriticalSection(Primes_CS),THEN(Assignment(PrimeCount,1),$0)))
3. CALL(Search) -> THEN(PARALLEL(Parameter(thread,0,minus(NUM_THREADS,1)),
   CALL(Search,Parameters(thread))), WAIT(ProcessedAll(NUM_THREADS)))
4. Search($0) -> Search(Parameters(ThreadNum),$0)
5. Assignment(start,1) -> Assignment(start,plus(multiply(ThreadNum,BLOCKSIZE),1))
6. Assignment(end,MAX_NUMBERS) ->
   Assignment(end,plus(multiply(ThreadNum,BLOCKSIZE),BLOCKSIZE))
7. IF($0,THEN(Assignment(ArrayElement(Primes,PrimeCount),number),
   Assignment(PrimeCount,plus(PrimeCount,1)))) ->
   IF($0,CriticalSection(THEN(Assignment(ArrayElement(Primes,PrimeCount),number),
   Assignment(PrimeCount,plus(PrimeCount,1)))))
8. THEN(UNTIL($0,$1),END(Search)) ->
   THEN(UNTIL($0,$1),THEN(ControlPoint(ThreadComplete),END(Search)))

Первое правило добавляет необходимые глобальные определения (см. пример 2). Правило 2 реализует равенство (3) – инициализация критической секции. Правила 3, 5, 6, 8 реализуют основное равенство (1) – асинхронное выполнение ветвей (правило 3), модификацию начального и конечного значений для каждой ветви (правила 5–6) и добавление контрольной точки в конце каждой ветви (правило 8). Правило 4 добавляет в функции Поиск параметр ThreadNum (см. пример 2). Правило 7 реализует равенство (2) – добавление критической секции.

Как видно из этого примера, по каждому равенству САА можно построить соответствующее правило (или набор правил) TermWare, что позволяет автоматизировать применение равенств.

Ввиду ограниченности объема работы другие правила трансформации не приводятся.

Заключение

В работе рассмотрены средства формализованного проектирования и трансформации последовательных и параллельных алгоритмов, базирующиеся на модифицированных САА. Алгоритмы представлены в естественно-лингвистической форме – в языке САА/1, осо-

бенностями которого являются простота в обучении и использовании, а также независимость от языка программирования и возможность автоматизированного перевода в произвольный язык для обеспечения адекватности алгоритмов и ассоциированных с ними программ. Оптимизация алгоритмов базируется на аппарате специальных равенств, –тождеств и соотношений, развитом в алгебрах алгоритмов. В работе показано, как для автоматизации применения таких равенств к схемам может быть применена система символьных вычислений TermWare, преимуществом использования которой является ее открытость. Кроме того, в процессе оптимизации программ использована система Thread Profiler, позволяющая идентифицировать фрагменты исходных текстов, нуждающиеся в доработке.

К перспективам развития разработанных инструментальных средств в рамках создания эффективных многопоточных программ следует отнести следующие:

- перенесение инструментария «ИПС» и TermWare на платформу MS .Net;
- дополнение существующего инструментария «ИПС» автоматизированными средствами оценки сложности проектируемых алгоритмов;
- создание базы данных представленных в алгеброалгоритмическом виде типичных шаблонов проектирования и оптимизирующих преобразований последовательных и параллельных алгоритмов и программ (включая использование Intel Thread Profiler и VTune) для их автоматизированного использования (на примере конкретной предметной области алгоритмов сортировки);
- создание на основе технологии переписывающих правил настраиваемых стратегий совместного использования средств «ИПС», TermWare и Intel Thread Profiler, позволяющих быстро идентифицировать фрагменты программного кода, критичные с точки зрения производительности, а также выбирать и применять оптимизирующие преобразования программ (на примере указанной предметной области).

1. *Г. Эндрюс*. Основы многопоточного, параллельного и распределенного программирования. – М.: ИД ”Вильямс”, 2003. – 505 с.
2. *Intel* Thread Profiler 3.0 for Windows. – http://www.intel.com/cd/software/products/asmo-na/eng/threading/286749.htm
3. *Intel*® VTune™ Performance Analyzers – Intel® Software Network http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm
4. *Яценко Е.А., Мохница А.С.* Инструментальные средства конструирования синтаксически правильных параллельных алгоритмов и программ // Проблемы программирования. – 2004. – № 2–3. – С. 444–450.
5. *Дорошенко А.Ю., Фінін Г.С., Цейтлін Г.О.* Алгеброалгоритмічні основи програмування. Об’єктна орієнтація і паралелізм. – К.: Наук. думка, 2004. – 458 с.
6. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications, Fundamenta Informaticae, 2006. – Vol. 72, N1–3. – P. 95–108.
7. *TermWare*. – http://www.gradsoft.com.ua/products/termware_rus.html
8. *Дорошенко А.Е., Шевченко Р.С.* Система символьных вычислений для программирования динамических приложений // Проблемы программирования. – 2005. – № 4. – С. 718–727.
9. *Intel* Thread Profiler for Windows. Getting Started Guide. – 2006.
10. *Цейтлин Г.Е.* Введение в алгоритмику. – Киев: Сфера, 1998. – 310 с.

Получено 15.01.2007

Об авторах:

Дорошенко Анатолий Ефимович,
доктор физ.-мат. наук, профессор,
заведующий отделом теории компьютерных вычислений,

Яценко Елена Анатольевна,
кандидат физ.-мат. наук,
научный сотрудник,

Жереб Константин Анатольевич,
аспирант физико-технического учебно-научного центра НАН Украины.

Место работы авторов:

Институт программных систем
НАН Украины,
проспект Академика Глушкова, 40.
03680, Киев-187, Украина.
тел. (044) 526 1538,
e-mail: dor@isofts.kiev.ua, aiyat@i.com.ua

Физико-технический учебно-научный
центр НАН Украины,
бульвар Вернадского, 36.
03142, Киев-142, Украина.
тел. (044) 424 3025,
e-mail: zhereb@gmail.com