УДК 681.3

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

ОБ ОЦЕНКЕ СЛОЖНОСТИ И КООРДИНАЦИИ ВЫЧИСЛЕНИЙ В МНОГОПОТОЧНЫХ ПРОГРАММАХ

Предложен метод оценки вычислительной сложности алгоритмов на основе алгебраического подхода. Разработана методика применения системы переписывания правил для преобразования конструкций координации вычислений в многопоточных программах. Приведены примеры трансформаций параллельных программ и результаты их выполнения на многоядерной архитектуре.

Введение

В работе [1] авторами было рассмотрено совместное использование разработанных алгеброалгоритмического инструментария проектирования и синтеза программ («ИПС») и системы переписывания правил TermWare для автоматизации трансформации многопоточных программ с общей памятью. Инструментарий «ИПС» базируется на диалоговом конструировании схем алгоритмов в системах алгоритмических алгебр (САА) и допускает синтез последовательных и параллельных программ в различных, в том числе, объектно-ориентированных языках программирования (Java, C++ и др.).

В данной работе предложен метод оценки сложности последовательных и параллельных алгоритмов, представленных САА-схемами. Разработана также методика применения TermWare для преобразования конструкций координации вычислений в многопоточных программах для многоядерных архитектур [2]. Координационный аспект параллельных вычислений, как известно [3], отражает свойства средств синхронизации и обменов между процессорами, которые могут существенно влиять на производительность параллельной системы. К средствам координации относятся классические средства (семафоры, критические секции, мониторы) и более сложные (конвейеры, архитектура клиент-сервер и др.), а также правила координации, которые устанавливают зависимости между координируемыми объектами (потоками или процессами) и средства реализации таких зависимостей. Правила также описывают переход от одних средств координации объектов к другим, более эффективным по тем или иным критериям. Автоматизированное применение таких правил к алгоритму или программе позволяет осуществлять система символьных вычислений TermWare. При этом для проектирования программы на более высоком уровне представления (например, в естественно-лингвистическом или граф-схемном виде [4]), применяется инструментарий «ИПС». Помимо стандартных средств синхронизации, реализованных в языках программирования, могут также использоваться специализированные средства, разработанные вручную, позволяющие повысить эффективность многопоточных программ. Отметим, что с задачей разработки эффективных алгоритмов непосредственно связана задача оценки их сложности.

Материал работы подчинен следующей структуре. В разделе 1 предложен метод получения оценок сложности для САА-схем алгоритмов. В разделе 2 рассматривается методика применения системы TermWare совместно с «ИПС» для разработки многопоточных программ. В разделе 3 использование данной системы проиллюстрировано на задаче определения простых чисел (Primes), анализ которой начат в [1], и программе моделирования броуновского движения (Brown), которая может быть отнесена к классу программ с интенсивной синхронизацией. Приведены результаты экспериментов по выполнению различных вариантов данных программ на многоядерной платформе.

1. Оценка вычислительной сложности схем алгоритмов

При анализе сложности алгоритма обычно осуществляется оценка порядка роста необходимых для решения задачи времени и емкости памяти при увеличении размера n входных данных. Напомним [5], что под временной сложностью $T(n)$ алгоритма понимается время, затрачиваемое алгоритмом, как функция размера задачи n. Поведение этой сложности в пределе при увеличении размера задачи называется асимптотической временной сложностью. Аналогично можно определить емкостную сложность (емкость памяти, требуемую для решения задачи как функцию размера задачи) и асимптотическую емкостную сложность. Отметим, что именно асимптотическая сложность алгоритма определяет в итоге размер задач, которые можно решить этим алгоритмом. Для описания асимптотической скорости роста функций используется O-символика. Далее приведены основные правила для определения асимптотической сложности [6]:

$O(k * f(n)) = O(f(n)),$

где $k \neq 0$ – константа;

$O(f(n) * g(n)) = O(f(n)) * O(g(n))$ или

$O(f(n) / g(n)) = O(f(n)) / O(g(n))$;

$O(f(n)) + O(g(n)) = O(\max\{|f(n)|, |g(n)|\}).$

Алгоритмы по их асимптотической вычислительной сложности могут быть классифицированы следующим образом:

- $O(1)$ – постоянные. Сложность таких алгоритмов не зависит от n (пример: проверка числа на четность или нечетность);
- $O(n)$ – линейные (например, алгоритм нахождения элемента в неотсортированном списке);
- $O(\log n)$ – логарифмические. Эта сложность встречается обычно в алгоритмах, которые разделяют задачу на подзадачи и решают их по отдельности (например, бинарный поиск);
- $O(n \log n)$ – квазилинейные. Такая сложность характерна для алгоритмов, которые разделяют задачу на подзадачи, а затем, решив их, соединяют полученные решения (сортировка методом Шелла, сортировка слиянием);
- $O(n^c)$ – полиномиальные, где $c > 1$ – константа. Примеры: $O(n^2)$ – квадратичная сложность (некоторые алгоритмы сортировки); $O(n^3)$ – кубическая сложность;
- $O(c^n)$ – экспоненциальные ($c > 1$); $O(n!)$ – факториальные. Такие алгоритмы чаще всего возникают в результате подхода, известного как “метод грубой силы” (например, полный перебор сочетаний элементов).

Временная сложность алгоритма может быть вычислена исходя из анализа его управляющих структур (таблица). В таблице конструкции алгоритма представлены в САА [3]; O(*условие*) и O(*оператор*) обозначают соответственно асимптотическую сложность для некоторого условия и оператора.

Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов. В общем случае существуют два способа анализа сложности алгоритма: восходящий (от внутренних управляющих структур к внешним) и нисходящий (от внешних к внутренним) [6]. Для оценки сложности параллельных алгоритмов предлагается вычислять общее количество операций, выполняемых в наиболее трудоемкой ветви.

Пример 1. Рассмотрим схему алгоритма:

```
"(i) := (1)"
ЗАТЕМ
ПОКА НЕ '(i) > (n)'
ЦИКЛ
    ...
    ЗАТЕМ
    "(i) := (i) + (1)"
КОНЕЦ ЦИКЛА
```

Таблица. Вычисление асимптотической временной сложности для конструкций алгоритма

Вид управляющей структуры	Сложность
Базисный оператор или базисное условие	$O(1)$
оператор1 ЗАТЕМ *оператор2*	$O(оператор1) + O(оператор2)$
ЕСЛИ *условие* ТО *оператор1* ИНАЧЕ *оператор2* КОНЕЦ ЕСЛИ	$\max\{O(условие), O(оператор1), O(оператор2)\}$
ПОКА НЕ *условие_окончания_цикла* ЦИКЛ *оператор* КОНЕЦ ЦИКЛА	$O(n) * O(оператор)$
оператор1 ПАРАЛЛЕЛЬНО *оператор2*	$\max\{O(оператор1), O(оператор2)\}$
ПАРАЛЛЕЛЬНО((*i*) = (1),..., (*n*)) (*оператор*)	$O(оператор)$
ЖДАТЬ *условие*	$O(условие)$
КТ *условие*	$O(условие)$

При условии, что сложность тела цикла – $O(1)$, сложность данного алгоритма будет $O(n)$.

Если один цикл вложен в другой и оба цикла зависят от размера одной и той же переменной (n), то вся конструкция характеризуется квадратичной сложностью $O(n^2)$:

```
"(i) := (1)"
ЗАТЕМ
ПОКА НЕ '(i) > (n)'
ЦИКЛ
    …
    "(j) := (1)"
    ЗАТЕМ
    ПОКА НЕ '(j) > (n)'
    ЦИКЛ
        …
        "(j) := (j) + (1)"
    КОНЕЦ ЦИКЛА
    …
    "(i) := (i) + (1)"
КОНЕЦ ЦИКЛА
```

При оценке сложности программ обычно рассматривается средний случай – ожидаемое время работы программы на “типичных” входных данных, и худший случай – ожидаемое время работы программы на самых “плохих” входных данных. Лучший, средний и худший случаи играют значительную роль в задачах сортировки [7].

Анализ асимптотической сложности получил широкое распространение во многих практических приложениях. Тем не менее, к основным недостаткам данного подхода можно отнести следующие [6]:

1) для сложных алгоритмов получение O-оценок, как правило, либо очень трудоемко, либо практически невозможно;

2) часто трудно определить сложность “в среднем”;

3) O-оценки являются достаточно грубыми для отображения более тонких отличий алгоритмов;

4) O-анализ дает слишком мало информации (или вовсе ее не дает) для анализа поведения алгоритмов при обработке небольших объемов данных.

Определение сложности в O-обозначениях – далеко не тривиальная задача. В частности, эффективность бинарного поиска определяется не глубиной вложенности циклов, а способом выбора каждой очередной попытки. Из-за трудностей, связанных с проведением анализа временной сложности алгоритма “в среднем”, часто приходится довольствоваться оценками для худшего и лучшего случаев.

Таким образом, оценка вычислительной сложности алгоритма, получаемая на основе рассмотренного метода, может быть не совсем точной. В этом случае дополнительно предлагается использовать способ получения оценки через вычисление количества выполняемых базисных операций при различных размерах n входных данных. При этом могут подсчитываться либо все базисные операции алго-

ритма, либо только существенные для данной задачи (например, для сортировок обычно вычисляется количество сравнений и перестановок элементов). Далее по результатам строится график временной сложности $T(n)$, анализ которого позволит определить порядок функции $T(n)$. На рис. 1 показаны примеры зависимостей $T(n)$ для последовательных алгоритмов сортировки массивов [7]: $T(n) = n^2 + n - 2$ (пузырьковая сортировка), $T(n) = (n^2 + 5n - 6)/2$ (челночная) и $T(n) = O(n \log n)$ (сортировка слиянием).

хранения данных некоторого алгоритма, могут использоваться величины $S = 1$ (для символьного типа), $Z_1 = 2$, $Z_2 = 4$ (для целочисленных типов); $R_1 = 4$, $R_2 = 8$, $R_3 = 10$ (для вещественных типов). Массив, состоящий из n элементов, имеет размер nX, где X – размер скаляра.

Пример 2. Для иллюстрации получения оценки временной сложности для параллельных вычислений, рассмотрим САА-схему алгоритма моделирования броуновского движения. Суть данного ал-

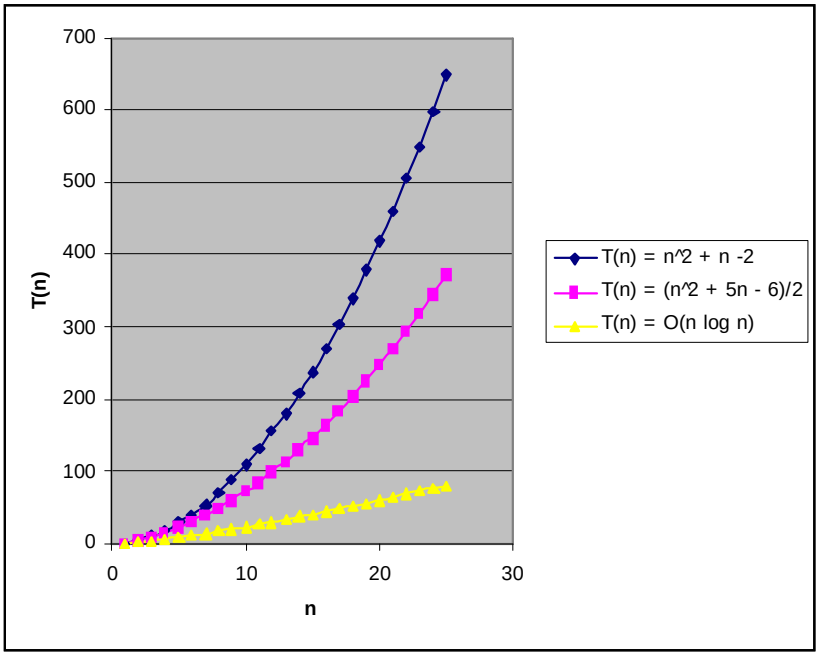

Рис. 1. Примеры графиков временной сложности $T(n)$

Для вычисления емкостной сложности алгоритма требуется проанализировать блоки глобального и локального объявлений переменных [1] схемы алгоритма с учетом типов их данных. Отметим, что определение емкости алгоритма тесно связано с условиями его реализации для конкретной ЭВМ и используемого языка программирования [8]. С другой стороны, набор типов данных, используемых в вычислениях, относительно постоянен. Среди скалярных величин это целочисленные типы, вещественные типы, строковые типы. Среди векторных величин это, в основном, массивы скаляров, объем которых определяется совокупным объемом соответствующих скаляров. Для вычисления объема памяти в байтах, необходимого для

горитма состоит в совместном функционировании `NUM_THREADS` параллельных потоков, каждый из которых моделирует движение некоторой частицы (примеси) в одномерном кристалле (массиве) `Crystal` длины `CrystalLen > 0`. Количество потоков равно количеству примесей: `NUM_THREADS = AdmixtureCount`. При инициализации программы все примеси находятся в первой ячейке кристалла. Одна итерация каждого потока заключается в переходе частицы в соседнюю (левую или правую) ячейку кристалла в зависимости от сгенерированного случайного числа. Новая позиция частицы вычисляется в операторе "`ОпределитьНовуюПозицию`". Сокращенная САА-схема для данного алгоритма имеет вид

```
СХЕМА БРОУНОВСКОЕ_ДВИЖЕНИЕ/П

"ОсновнойСоставнойОператор"
==== "Инициализация данных"
        ЗАТЕМ
        ПАРАЛЛЕЛЬНО((ThreadNum) = (0), ..., (NUM_THREADS - 1))
        (
           "ДвижениеЧастицы(ThreadNum)"
        )
        ЗАТЕМ
        ЖДАТЬ 'Обработка во всех (NUM_THREADS) ветвях закончена'

"ОпределитьНовуюПозицию(right_margin, pos, limit, ThreadNum)"
==== "(res) := (pos)"
        ЗАТЕМ
        "Сгенерировать случайное число (r) из отрезка [0..1]"
        ЗАТЕМ
        ЕСЛИ '(r) >= (limit)' И НЕ('(pos) = (right_margin)')
        ТО "(res) := (pos + 1)"
        КОНЕЦ ЕСЛИ
        ЗАТЕМ
        ЕСЛИ '(r) < (limit)' И НЕ('(pos) = (0)')
        ТО "(res) := (pos - 1)"
        КОНЕЦ ЕСЛИ

"ДвижениеЧастицы(ThreadNum)"
==== "Инициализировать генератор случайных чисел"
        ЗАТЕМ
        ДЛЯ '(iteration_counter) от (1) до (IterationAmount)'
        ЦИКЛ
             "(nextx) = ОпределитьНовуюПозицию(CrystalLen - 1, x, ProbabilityLimit, ThreadNum)"
             ЗАТЕМ
             ЕСЛИ НЕ('(nextx) = (x)')
             ТО КритическаяСекция(Admixture_CS)
                (
                   "Уменьшить (Crystal[x]) на (1)"
                   ЗАТЕМ
                   "Увеличить (Crystal[nextx]) на (1)"
                )
                ЗАТЕМ
                "(x) := (nextx)"
             КОНЕЦ ЕСЛИ
        КОНЕЦ ЦИКЛА
КОНЕЦ СХЕМЫ
```

Рассмотрим процесс вычисления для приведенного алгоритма асимптотической временной сложности $T(n)$, при n равном количеству итераций в каждом потоке (`IterationAmount`). Выполним восходящий анализ управляющих структур алгоритма (от внутренних к внешним) в соответствии с правилами из таблицы.

Представим последовательность вычислений сложности конструкций составного оператора “`ДвижениеЧастицы(ThreadNum)`” следующими формулами:

$$O(cs) = O(1) + O(1) = O(1);$$
$$O(if) = \max\{O(1), O(cs) + O(1)\} = \max\{O(1), O(1) + O(1)\} = O(1);$$
$$O(cycle) = O(n) \cdot (O(1) + O(if)) = O(n) \cdot (O(1) + O(1)) = O(n);$$
$$O(co) = O(1) + O(cycle) = O(1) + O(n) = O(n),$$

где $O(cs)$, $O(if)$, $O(cycle)$, $O(co)$ – асимптотическая сложность для критической секции, условного оператора, цикла и составного оператора “`ДвижениеЧастицы(ThreadNum)`” соответственно.

Далее приведена последовательность вычислений сложности для основного составного оператора алгоритма:

$$O(par) = O(co) = O(n);$$
$$O(algo) = O(1) + O(par) + O(1) = O(1) + O(n) + O(1) = O(n),$$

где $O(par)$, $O(algo)$ – асимптотическая сложность для параллельной конструкции и всего алгоритма соответственно. Таким образом, рассмотренный параллельный алгоритм имеет линейную сложность $O(n)$. Отметим, что сложность соответствующего ему последовательного алгоритма составляет $O(n^2)$.

2. Методика применения системы TermWare совместно с инструментарием «ИПС»

Как уже упоминалось [1], использование системы TermWare позволяет дополнить «ИПС» возможностью автоматизированного выполнения преобразований. Поскольку система TermWare [9–11] является достаточно гибкой, возможны различные варианты ее применения, перечисленные далее.

1. Создание правил для автоматизации конкретного преобразования. Такие правила позволяют быстро осуществлять достаточно сложные преобразования. При этом правила могут как вводиться вручную (в текстовой или визуальной форме), так и генерироваться автоматически на основании действий пользователя. Во втором случае пользователь вручную осуществляет некоторое преобразование схемы алгоритма, при этом отмечая начало и конец трансформации. На основании этих данных система автоматически создает правила, переводящие начальный терм в конечный (эта возможность аналогична записи макроса в современных средах разработки; при этом «макрос» представляется в декларативном виде). Полученные правила будут соответствовать преобразованию конкретной программы; однако есть возможность редактировать эти правила, в частности, заменяя отдельные подтермы на переменные. При этом пользователь указывает терм, конкретное значение которого не существенно для корректности правила; все вхождения этого терма автоматически заменяются переменной TermWare с уникальным названием. Данная возможность позволяет пользователям, незнакомым с системой TermWare, использовать возможности автоматизации, а также изучать TermWare на конкретных примерах.

2. Восстановление САА-схемы из исходного кода. Для этого используется парсер целевого языка (C/C++, Java, C# и др.), который преобразовывает исходный код в терм специального вида, соответствующий дереву синтаксического разбора. После этого происходит автоматизированное преобразование полученного терма в терм, соответствующий схеме алгоритма. Для каждого элемента схемы задаются правила, которые обнаруживают данный элемент в дереве синтаксического разбора. После применения этих правил, система проверяет наличие в полученном терме элементов, которые не могут быть частью САА-схемы (т.е. конструкций, не предусмотренных в правилах). Пользователь вручную задает правила обработки этих элементов (например, они могут игнорироваться или сохраняться в виде специальных элементов схемы, не предназначенных для дальнейшей модификации). Таким образом, пользователь может начать работу с «ИПС», не изучая детально правила создания схем.

3. Преобразования конструкций, характерных для определенной платформы (языка). Многие элементы схем (в особенности конструкции координации многопоточных программ) имеют похожую семантику, но различный синтаксис в разных языках или при использовании разных библиотек. Возможно создание правил, осуществляющих преобразования между такими конструкциями. Это позволяет облегчить переход от одной платформы к другой.

4. Применение преобразований, ориентированных на определенные классы задач. Разработка преобразований, которые автоматически оптимизируют любую программу, представляется практически невозможной. Тем не менее, существуют определенные классы задач, для которых можно построить оптимизирующие преобразования, применимые ко всем задачам данного класса. Поэтому возможно создание библиотеки таких преобразований, при этом пользователю предоставляется возможность выбора конкретной системы

правил. Даже если общие правила неприменимы к данной задаче, они могут использоваться в качестве основы для написания специализированных правил. Возможно также создание правил, определяющих возможность применения данного преобразования.

Основой функционирования системы TermWare в рамках «ИПС» является создание и применение систем правил. Каждая система правил соответствует некоторому преобразованию схемы алгоритма. Кроме самих правил TermWare, система правил может включать дополнительную информацию, например, описание преобразования или ссылки на другие системы правил, которые могут применяться совместно с данной. Возможны также информационные системы правил, которые не осуществляют преобразование схемы, а вычисляют некоторые ее характеристики.

Кроме работы с системами правил, TermWare позволяет выполнять и другие действия. Например, уже упоминалась возможность автоматического создания правил, исходя из начального и конечного терма, а также модификации правил с помощью замены подтерма на переменную. Есть возможность отслеживания определенных термов; например, при преобразовании в терм, который должен соответствовать некоторым требованиям (быть корректной САА-схемой, соответствовать конкретной платформе) система указывает пользователю на не преобразованные термы (которые не удовлетворяют требованиям). Можно также использовать так называемые TODO-термы. Эти термы имеют вид TODO_TermName и представляют терм вида TermName, который может требовать определенной доработки. TODO-термы отслеживаются в двух случаях: они выводятся как список задач, стоящих перед пользователем (аналогично использованию комментариев, содержащих TODO, в современных средствах разработки). Кроме того, если элемент TermName является допустимым в конечном терме, возможно автоматическое создание правил TODO_TermName → TermName.

3. Примеры использования системы TermWare

3.1. Пример: задача нахождения простых чисел (Primes). В качестве примера использования TermWare рассмотрим задачу определения общего количества простых чисел от 1 до некоторого числа [1]. Вычисления в ней осуществляются путем проверки каждого нечетного числа, является ли оно нацело делимым на меньшие нечетные факторы. Данная задача принадлежит к классу задач, в которых осуществляются некоторые независимые вычисления, после чего результат вычислений сохраняется в общую память. Такие алгоритмы достаточно легко поддаются распараллеливанию, однако при этом есть некоторые особенности, влияющие на производительность многопоточной программы. В качестве исходной схемы для данной задачи в [1] рассмотрена последовательная программа (Sequential), которая была преобразована в параллельную путем разбиения главного цикла (в котором перебираются числа – кандидаты в простые) на `NUM_THREADS` циклов (где `NUM_THREADS` – количество потоков). В простейшем варианте программы (BlockTasksThread) каждый поток обрабатывает последовательные числа. Эта программа позволяет использовать аппаратные средства многоядерных систем, но недостаточно эффективно. Для повышения производительности вычисления в каждом потоке модифицируются таким образом, чтобы все потоки выполняли примерно одинаковый объем работ; при этом получается достаточно эффективная программа InterleavedTaskThread. Следующим шагом является улучшение программы за счет использования функций типа `Interlocked`. При этом была получена еще более эффективная программа InterlockedThread, хотя изменение производительности по сравнению с предыдущей программой InterleavedTaskThread оказывается незначительным. Перечисленные преобразования были применены к алгоритмам с использованием правил в системе TermWare.

На рис. 2 и 3 показаны результаты сравнения производительности полученных программ, выполненных на процессоре Intel Core Duo. Приведены зависимости времени исполнения от количества чисел, а также от количества потоков.

Как видно из графиков, простейший способ распараллеливания (BlockTasksThread) позволяет использовать аппаратные ресурсы двухядерной платформы, но не полностью. Использование более правильной схемы распараллеливания позволяет повысить производительность практически вдвое по сравнению с последовательным случаем, т.е. полностью использовать аппаратные ресурсы. Заметим также, что при использовании большего количества потоков производительность реализации BlockTasksThread возрастает, приближаясь к более эффективным InterleavedTaskThread и InterlockedThread; таким образом, при использовании более мелких задач правильное распределение вычислений между ними становится не столь существенным.

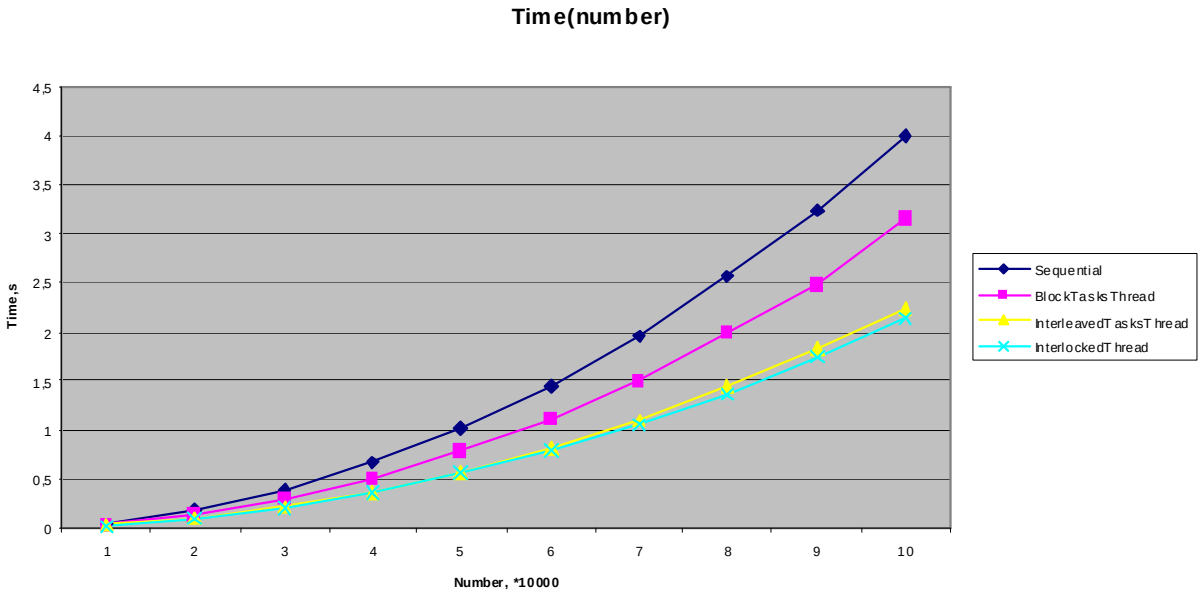

Рис. 2. Зависимость времени исполнения от количества чисел (2 потока)

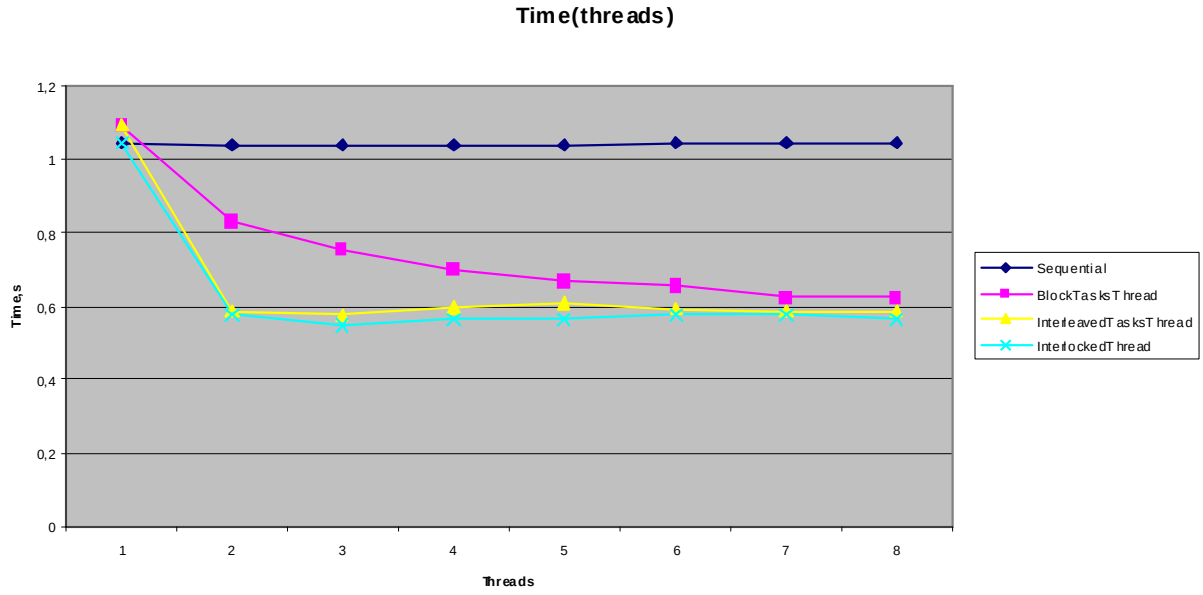

Рис. 3. Зависимость времени исполнения от количества потоков (50000 чисел)

3.2. Пример: задача Brown. В качестве примера применения алгебраического подхода для преобразования средств координации вычислений рассмотрим многопоточную программу моделирования броуновского движения, реализованную по схеме алгоритма из примера 2 (см. раздел 1).

Особенностью данной задачи является то, что большая часть времени в ней расходуется не на вычисления, а на синхронизации потоков. Поэтому простые способы реализации оказываются малоэффективными. Например, исходная реализация задачи (программа SmallLockThread, которая реализована с использованием критической секции) на двуядерной системе работает медленнее, чем последовательная реализация (Sequential).

На примере этой задачи можно продемонстрировать использование различных типов преобразований, не меняющих вычислительной части программы, но затрагивающих координационный аспект. Первым типом преобразования является изменение критической области. Возможно либо расширение критической области (включение в нее дополнительных операторов), либо разбиение критической области на несколько меньших критических областей. В первом случае преобразования осуществляются автоматически, во втором от разработчика требуется указать идентификаторы новых критических областей. В частности, в данной задаче имеет смысл использовать массив критических секций и блокировать каждую операцию с помощью критической секции, соответствующей номеру изменяемой ячейки кристалла. Проиллюстрируем упомянутые преобразования.

Рассмотренному в примере 2 составному оператору "ДвижениеЧастицы(ThreadNum)", реализующему в схеме БРОУНОВСКОЕ_ДВИЖЕНИЕ/П отдельный поток, соответствует следующий терм

```
MoveParticle(
   Parameters(ThreadNum),
    THEN(InitRandom,
     THEN(
      FOR(Parameters(iteration_counter, 1, IterationAmount),
       THEN(
        Assignment(nextx, CalculateNewPosition(minus(CrystalLen, 1), x,
                    ProbabilityLimit, ThreadNum)),
        THEN(
         IF(logical_not(eq(nextx, x)),
          THEN(
           CriticalSection(Admixture_CS,
            THEN(
             Decrement(ArrayElement(Crystal, x)),
             THEN(
              Increment(ArrayElement(Crystal, nextx)), NIL))),
           THEN(
            Assignment(x, nextx), NIL))), NIL))), NIL))).
```

Расширение критической секции использует правила следующего вида (система правил R_1):

1. THEN(CriticalSection($x0, $x1), $x2) → CriticalSection($x0, THEN($x1, $x2)).
2. THEN($x0, CriticalSection($x1, $x2)) → CriticalSection($x1, THEN($x0, $x2)).
3. IF($x0, CriticalSection($x1, $x2)) → CriticalSection($x1, IF($x0, $x2)).

системы TermWare:

Первые два правила определяют расширение критической секции по линейным участкам кода; третье определяет включение в критическую секцию конструкций `if` (подобные правила можно создать и для других управляющих конструкций). В результате применения этих правил исходный терм преобразуется в терм, соответствующий реализации потока в новой программе BigLockThread:

```
MoveParticle(
 Parameters(ThreadNum),
  THEN(InitRandom,
   THEN(
    FOR(Parameters(iteration_counter, 1, IterationAmount),
     CriticalSection(Admixture_CS,
      THEN(
       Assignment(nextx, CalculateNewPosition( minus(CrystalLen,1), x,
                  ProbabilityLimit, ThreadNum)),
       THEN(
        IF(
         logical_not(eq(nextx, x)),
         THEN(
          THEN(
           Decrement(ArrayElement(Crystal, x)),
           THEN(
            Increment(ArrayElement(Crystal, nextx)), NIL)),
          THEN(
           Assignment(x, nextx), NIL))), NIL)))), NIL))).
```

Разбиение критической секции на несколько меньших критических секций осуществляется с помощью следующих правил (R_2):

Далее в алгоритме можно использовать массив критических секций и блокировать каждую операцию с помощью критической секции, соответствующей номеру из-

1. CriticalSection($name, THEN($x, $y)) →
   THEN(CriticalSection1(TODO_Name($name, 0),$x),
   CriticalSection1(TODO_Name($name, 1), $y)).
2. CriticalSection1(TODO_Name($name, $n),THEN($x, $y)) →
   THEN(CriticalSection1(TODO_Name($name, $n), $x),
   CriticalSection1(TODO_Name($name,$n+1),$y)).
3. CriticalSection1($name, NIL) → NIL.

Первое правило выделяет первый оператор из критической секции в отдельную

меняемой ячейки кристалла. В этом случае применяются правила (R_3):

1. TODO_Name(Admixture_CS, 0) → ArrayElement(Admixture_CS, x).
2. TODO_Name(Admixture_CS, 1) → ArrayElement(Admixture_CS, nextx).
3. CriticalSection1($x, $y) → CriticalSection($x, $y).

Первые два правила определяют конкретные названия критических секций. Правило 3 переводит терм со специальным названием `CriticalSection1` в стандартный терм `CriticalSection`.

Заметим, что в общем случае эти преобразования скорее ухудшают свойства программы. Расширение критических секций (программа BigLockThread) уменьшает количество кода, который может выполняться параллельно (хотя упрощение структуры программы может привести к использованию оптимизаций компилятора, что улучшает производительность). Разбиение критической секции (программа SeparatedLockThread) повышает накладные расходы на синхронизацию; кроме того, полученная программа оказывается не полностью эквивалентной исходной. При разбиении критической секции отдельные операторы выполняются атомарным образом, но между двумя последовательными операторами одного потока могут выполняться операторы другого (хотя в данной задаче это несущественно). Однако применение таких трансформаций позволяет вводить дополнительные средства синхронизации, которые могут быть более эффективными. Например, программа SeparatedLockThread содержит только операторы увеличения и уменьшения переменной, поэтому критические секции в ней можно заменить функциями `InterlockedDecrement` и `InterlockedIncrement`. Эта программа (InterlockedThread) является более эффективной, чем исходная (хотя и не строго эквивалентна ей, как и SeparatedLockThread).

Программа BigLockThread также может использоваться в качестве исходной при введении новых средств синхронизации. Однако при этом используются не стандартные средства синхронизации, а специально разработанный класс `LoopManipulation`. Этот класс предоставляет один метод `RunSafeLoop`, который принимает в качестве параметра тело цикла и количество итераций, и выполняет необходимое количество итераций в пределах критической области (фактически этод метод заменяет цикл `for` с вложенной критической секцией). При этом количество блокировок не равняется количеству итераций, а определяется специальным параметром класса (т.е. несколько итераций выполняются в пределах одной критической секции). За счет уменьшения количества критических секций существенно снижаются накладные расходы на синхронизацию, поэтому такая программа (LoopManipulationThread) является более эффективной. Заметим, что фактически данный класс является альтернативной реализацией алгебраической конструкции – цикла. Это значит, что хоть исходный код двух вариантов отличается достаточно существенно, но их алгебраическое представление оказывается практически одинаковым.

Также возможен переход к использованию специализированных библиотек синхронизации, таких как Concurrency and Coordination Runtime (CCR) [12, 13] и C# Software Transactional Memory (SXM) [14, 15]. Библиотека CCR представляет собой набор программных конструкций для координации взаимодействий между потоками в C# [12]. Данные конструкции могут комбинироваться для представления сложных и гибких шаблонов взаимодействия (communication patterns). Библиотека CCR базируется на программирования на основе портов (port-based programming) и передачи сообщений, как механизме структурирования программного обеспечения и средстве обеспечения изолированности между различными компонентами для надежности и повышения степени параллелизма программы. Основным достоинством библиотеки CCR является возможность высокоуровневой координации задач, что исключает необходимость использования низкоуровневых конструкций синхронизации, таких как блокировки.

Software Transactional Memory (SXM) [14, 15] представляет собой API для

многопоточных вычислений, в которых разделяемые потоками данные синхронизируются без использования блокировок. Потоки синхронизируются с помощью транзакций памяти (memory transactions) – кратковременных вычислений, которые либо выполняются полностью (оказывают воздействие на данные) либо терпят неудачу и прерываются (не оказывают воздействия). С помощью SXM в программах указываются атомарные (atomic) блоки, упрощающие написание параллельных программ. Блок кода обозначается как атомарный, после чего компилятор и система поддержки выполнения гарантируют, что операции в пределах блока, включая вызовы функций, будут атомарными.

Для перехода к использованию специализированных библиотек координации возможно использование простых и достаточно общих алгебраических равенств (правил TermWare). При этом определенные конструкции синхронизации преобразуются в их аналоги в этих библиотеках; далее с помощью генератора кода создаются программы, использующие данные библиотеки (HandleCCR и SXMTransactional). Как и в случае программы LoopManipulationThread, несмотря на значительное различие кода на языке C#, алгебраические представления программ оказываются весьма близкими, и отражают существенное изменение – переход к новой реализации определенных операторов, а не детали реализации. Особенно это заметно для SXM, поскольку в этой системе вместо простого блока atomic{} используются довольно сложные правила вызова API (поскольку эта библиотека разрабатывалась в расчете на обычный компилятор C#, в ней не было возможности непосредственной реализации новых конструкций языка). Итак, алгебраическая запись позволяет скрыть технические детали реализации таких библиотек, показывая только существенные отличия. Кроме того, использование автоматически применяемых правил позволяет осуществить быстрый переход к использованию подобных библиотек, при этом нет необходимости вручную переписывать существующий код.

Полученные до настоящего времени программы использовали общие правила, что позволяло применять их к широкому классу задач. Однако при этом производительность таких программ оказывалась невысокой. Для того чтобы получить более существенный прирост производительности, необходимо использовать особенности задачи. Основной проблемой при создании эффективной параллельной реализации задачи моделирования броуновского движения является необходимость выполнения большого количества простых итераций. Для повышения производительности необходимо уменьшить количество синхронизаций (аналогично LoopManipulationThread), при этом не расширяя критическую область. Один из вариантов реализации данного подхода заключается в сохранении состояния каждой итерации в пределах соответствующего потока, с записью этого состояния в общую память после определенного количества итераций. Такая программа (InvisibleMovesThread) может быть получена с использованием системы правил, реализующей общую схему такого преобразования; при этом конкретные особенности (например, как происходит сохранение и обновление состояния) реализуются с помощью специальной системы правил, разработанной для данной задачи. При этом получается очень эффективная программа, производительность которой на двухядерной системе почти в 2 раза выше, чем у последовательной программы.

На рис. 4 и 5 показаны графики зависимости времени работы от количества итераций и количества частиц (потоков).

Как видно из приведенных графиков, большинство средств синхронизации оказывается не очень эффективным для данного класса задач. Существенное улучшение по сравнению с последовательной программой достигается только на программе InvisibleMovesThread. Программа LoopManipulationThread выполняется со скоростью, близкой к последовательной программе; все остальные программы оказываются неэффективными. Заметим, что программа на основе транзак

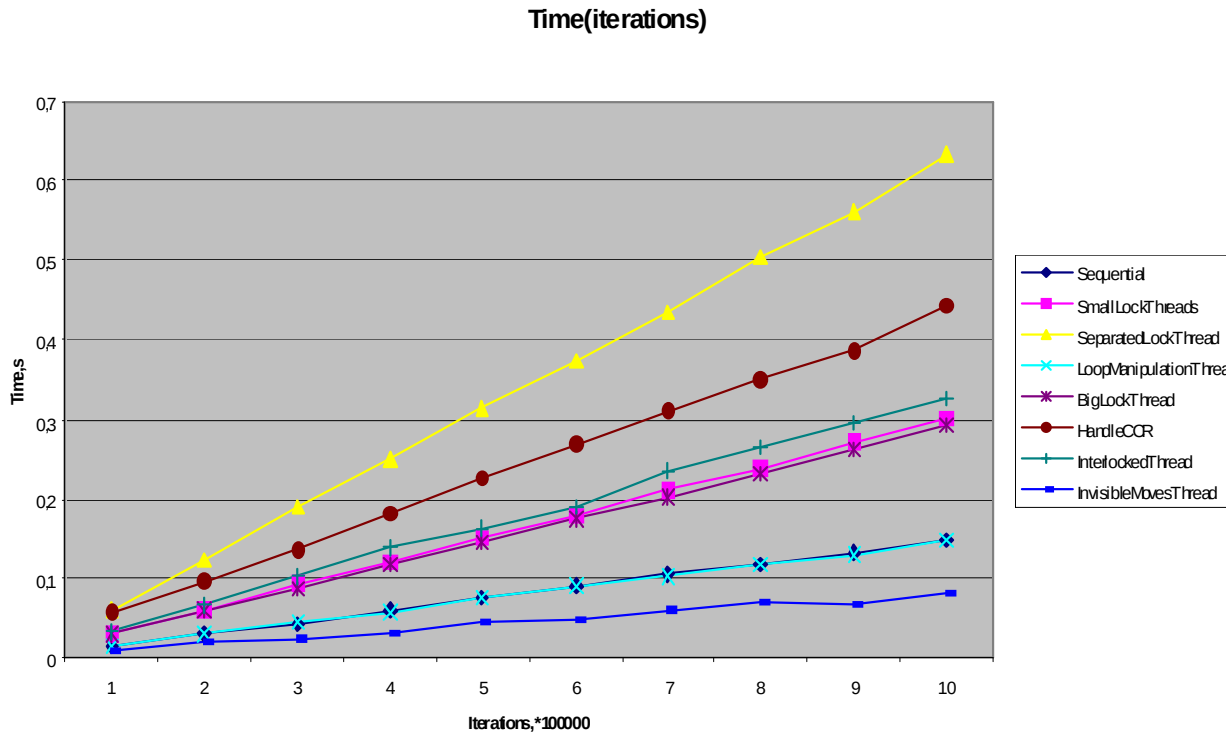

Рис. 4. Зависимость времени исполнения от количества итераций (2 частицы)

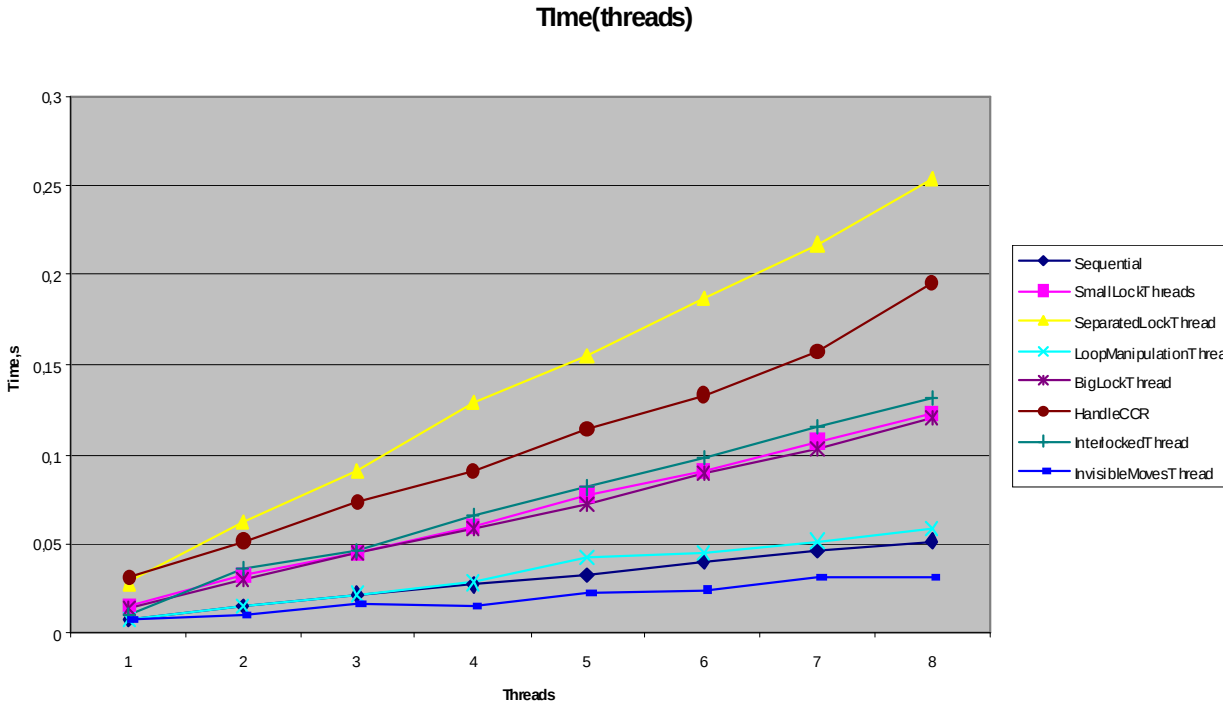

Рис. 5. Зависимость времени исполнения от количества частиц (100000 итераций)

ционной памяти не показана на рис. 4 и 5, поскольку время исполнения в этом случае существенно выше (например, для 2 частиц и 500000 итераций последовательная программа выполняется 0,09 с, худшая из приведенных программ – SeparatedLockThread – выполняется 0,39 с, а программа на основе транзакционной памяти – 8,31 с). Таким образом, подходы к синхронизации, претендующие на уни-

версальность, могут оказаться неэффективными для конкретных классов задач.

Заключение

В работе исследован вопрос получения оценок вычислительной сложности алгоритмов на основе алгебраического подхода. Предложена методика совместного использования интегрированного инструментария и системы TermWare для автоматизации применения преобразований к алгоритмам (в частности, направленных на оптимизацию программ). Приведенные примеры (Primes и Brown) демонстрируют возможности TermWare для перехода между различными средствами координации вычислений в многопоточных программах. Результаты экспериментов по выполнению полученных оптимизированных программ (InterleavedTaskThread, InterlockedThread и InvisibleMovesThread) на микропроцессоре с многоядерной архитектурой демонстрируют хорошую степень распараллеливаемости и масштабируемости вычислений.

Дальнейшее развитие разрабатываемых инструментальных средств будет также включать создание:

- трансформатора алгоритмов, интегрирующего TermWare и «ИПС», и предназначенного для конструирования и применения правил;
- средств облегченной синхронизации в многопоточных программах, улучшающих существующие методы и позволяющих достичь высокого качества многоядерных программ, интенсивно использующих синхронизацию.

1. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Формализованное проектирование эффективных многопоточных программ // Проблемы программирования. – 2007. – № 1. – С. 17–30.
2. *Shameem Akhter, Jason Roberts*. Multi-Core Programming. Increasing Performance through Software Multi-threading. – Intel Press, 2006. – 336 p.
3. *Дорошенко А.Ю., Фінін Г.С., Цейтлін Г.О.* Алгеброалгоритмічні основи програмування. – К.: Наук. думка, 2004. – 458 с.
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. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
6. *Структуры* и алгоритмы. Оценка программ. – http://www.structur.h1.ru/ocenka.htm
7. *Цейтлин Г.Е.* Введение в алгоритмику. – Киев: Сфера, 1998. – 310 с.
8. *Эффективные* алгоритмы. Теория и практика применения. Этапы полного построения алгоритма. – http://www.tspu.tula.ru/ivt/umr/ealg/docs/doc02/doc02.htm
9. *TermWare*. – http://www.gradsoft.com.ua/products/termware_rus.html
10. *Shevchenko R.* TermWare: Semantics Description, GradSoft Ltd, Kiev, Ukraine. – http://www.gradsoft.com.ua/rus/Products/termware/docs/Semantics_rus/index.html
11. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications, Fundamenta Informaticae, 2006. – Vol. 72, N 1–3. – P. 95–108.
12. *Chrysanthakopoulos G., Singh S.* An Asynchronous Messaging Library for C#. – http://research.microsoft.com/~tharris/scool/papers/sing.pdf
13. *An Asynchronous* Messaging Library for C# 2.0. – http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime
14. *C# Software* Transactional Memory. – http://research.microsoft.com/research/downloads/Details/6cfc842d-1c16-4739-afaf-edb35f544384/Details.aspx?CategoryID=
15. *Harris T., Jones S.P.* Transactional memory with data invariants. – http://research.microsoft.com/~tharris/papers/2006-transact.pdf

Получено 05.04.2007

Об авторах:

Дорошенко Анатолий Ефимович,
доктор физико-математических наук,
профессор,заведующий отделом теории компьютерных вычислений,

Яценко Елена Анатольевна,
кандидат физико-математических наук,
научный сотрудник,

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

Место работы авторов:

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

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