ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №4(27), 2015, с. 177–195

УДК 519.685.1+519.682.2

А. И. Адамович

Струи как основа реализации понятия Т-процесса для платформы JVM

Аннотация. Распространение и доступность современных параллельных аппаратно-программных платформ демонстрирует отставание уровня инструментов разработки параллельных приложений от нужд разработчиков программ. В ИПС РАН ведется разработка подхода к распараллеливанию программ, основанного на использовании модели вычислений «самотрансформация вычисляемой сети». В данной работе рассматриваются различные варианты подходов к реализации для платформы JVM понятия «Т-процесс» — базового понятия данной модели вычислений. Анализируются потенциальные проблемы, связанные с реализацией понятия «Т-процесс», как на основе классических потоков ОС/JDK, так и в случае внесения поддержки легковесных потоков непосредственно в код виртуальной машины. Предлагается подход к реализации Т-процессов, основанный на использовании понятия струй, т.е. легковесных потоков, реализуемых вне ядра JVM. Приводятся результаты экспериментального сравнения подходов к реализации понятия «Т-процесс», основанных на использовании классических потоков и струй (англ. fibers). Анализируется эффект от использования струй для реализации модели вычислений «самотрансформация вычисляемой сети», используемой в разрабатываемом языке параллельного программирования ajl для платформы JVM.

Ключевые слова и фразы: реализация языков программирования, параллельные вычисления, платформа JVM, автоматическое динамическое распараллеливание, потоки, струи.

Введение

Широкое распространение и доступность современных параллельных аппаратно-программных платформ в области вычислительных средств повседневного использования — таких, как снабженные многоядерными процессорами настольные и планшетные компьютеры,

Исследование выполнено в рамках НИР «Методы и средства разработки эффективного программного обеспечения, ориентированные на вычислительные системы, построенные на основе микропроцессоров с многоядерной архитектурой, и формальные основы высокоуровневых языков программирования для суперкомпьютеров с гибридной архитектурой» (№ г/р 01201455360) (госзадание ФАНО России).

а также мобильные телефоны — наглядно демонстрирует отставание уровня средств разработки параллельных приложений от нужд разработчиков программ. Потребитель может недорого приобрести планшетный компьютер с 8-ядерным процессором, и в то же время разработчики все еще не имеют возможности разрабатывать приложения, использующие предоставляемую такой аппаратурой вычислительную мощность, без значительных затрат дополнительных усилий как на саму разработку, так и на освоение соответствующих программных средств.

Для того, чтобы соответствовать современному уровню аппаратуры и требованиям разработчиков приложений, инструментальные программные средства должны обеспечивать автоматизированное (лучше — автоматическое) распараллеливание программ и, по возможности, самую высокую эффективность использования предоставляемой вычислительной мощности.

В ИПС РАН в течение значительного периода времени ведется разработка подхода к распараллеливанию программ, основанного на использовании оригинальной модели вычислений «самотрансформация вычисляемой сети» [1, 2]. Данный подход позволяет реализовать парадигму «автоматическое динамическое распараллеливание программ», что не только облегчает разработку параллельных прикладных программ, но и повышает качество распараллеливания. Действительно, если решение о передаче участка кода на выполнение тому или иному процессорному ядру принимается в динамике, то — за счет автоматического выявления простаивающих процессорных ядер и обеспечения их вычислительной нагрузкой — может достигаться более высокое значение коэффициента ускорения.

В настоящее время на упомянутых выше параллельных аппаратно-программных платформах повседневного использования наиболее часто и широко применяются базовые программных среды, ориентированные на язык Java. Поэтому естественным решением стало исследовать возможности эффективной реализации модели вычислений «самотрансформация вычисляемой сети» в одной из таких сред.

В качестве базовой среды для экспериментальной реализации была выбрана «классическая» платформа JVM, как обладающая наиболее развитым набором инструментария и наиболее доступная и удобная для проведения исследования.

Далее в статье обсуждаются модель вычисления «самотрансформация вычисляемой сети» и опыт реализации понятия Т-процесса — элемента данной модели вычислений — в рамках Т-системы, являющейся первой параллельной реализацией этой вычислительной модели. Затем анализируются различные подходы к реализации легковесных

потоков — основы для реализации понятия Т-процесс, и предлагается новый подход к реализации легковесных потоков на основе струй (англ. fibers). После этого обсуждается реализация поддержки понятия Т-процесса в экспериментальном языке программирования ajl. Наконец, приводятся и анализируются данные экспериментального сравнения реализации понятия Т-процесса на основе потоков и струй. В заключение статьи приводится общий обзор обсужденных тем и приведенных результатов, а также делается вывод о целесообразности использования струй для реализации понятия Т-процесса в языке ajl.

1. Проблема реализации понятия Т-процесса

1.1. Модель вычислений «самотрансформация вычисляемой сети»

Модель организации вычислений «самотрансформация вычислительной сети» основывается на следующих базовых принципах:

- В качестве основной парадигмы рассматривается функциональное программирование. Программа представляет собой набор чистых, без побочных эффектов, функций (Т-функций). Каждая Т-функция может иметь несколько аргументов и несколько результатов. В то же время тела Т-функций могут быть описаны в императивном стиле (на языках типа FORTRAN, C и т. п.) и могут включать в себя, в том числе, вызовы обычных, регулярных, функций входного языка. Важно только, чтобы:
  - всю информацию извне Т-функция получала только через свои аргументы;
  - вся информация из Т-функции передавалась только через явно описанные результаты.
- Вызов Т-функции `G`

```
cT –
1    [x,y,z] = G(a,b)
```

, производимый в процессе вычисления Т-функции `F`, выполняется нетрадиционным способом (т.н. сетевой вызов функции). При этом порождается новый Т-процесс с несколькими входами, в соответствии с числом аргументов Т-функции `G`, и несколькими выходами, в соответствии с числом результатов Т-функции `G`. Выходы нового Т-процесса связываются с соответствующими переменными Т-процесса `F` (Т-переменными или внешними переменными) — в нашем примере это Т-переменные `x,y,z` — отношением “поставщик-потребитель”, и тем самым Т-переменные-потребители

принимают неготовые (не вычисленные) значения. Порожденный Т-процесс G должен вычислить Т-функцию G и заменить неготовые значения у всех своих потребителей на соответствующие результаты Т-функции G.

- Все Т-процессы, вычисляющие Т-функции, порождаются в состоянии "готов к вычислению". Так, например, описанный выше вызов Т-функции G из Т-процесса F не ведет к потере Т-процессом F готовности, т.е. способности выполняться. Единственное событие, приводящее к потере готовности (к засыпанию) Т-процесса — это попытка Т-процесса выполнить с неготовым значением любую операцию, отличающуюся от операций передачи значений; операции передачи неготовых значений не приводят к потере процессом готовности и реализуются за счет изменения вычисляемой сети.
- Если некоторый Т-процесс предпринял попытку выполнить с неготовым значением операцию, отличную от операции передачи данных, то такой Т-процесс теряет готовность — засыпает (приостанавливается). Готовность Т-процесса будет восстановлена — Т-процесс будет разбужен — в тот момент, когда соответствующее неготовое значение (причина засыпания) будет заменено поставщиком на обычное значение.
- Таким образом, в каждый момент времени состояние вычисления представлено вычисляемой сетью, узлами которой являются запущенные Т-процессы и структуры данных, а дугами — отношения поставщик-потребитель. В процессе выполнения программы вычисляемая сеть самотрансформируется: в моменты вызовов Т-функций в ней появляются новые узлы, в моменты завершения вычисления всех потребляемых результатов Т-функций узлы исчезают, при выполнении допустимых операции с неготовыми значениями появляются и исчезают дуги.

Наличие в вычисляемой сети достаточного количества готовых к выполнению Т-процессов позволяет выполнять их параллельно.

1.2. Опыт реализации понятия Т-процесс в Т-системе

Из приведенного описания модели организации вычислений видно, что при ее реализации одной из основных базовых сущностей является Т-процесс. Т-процесс — это часть процесса вычислений, он служит для вычисления соответствующей ему Т-функции и может быть создан, передан на выполнение (запущен), приостановлен. Он может

возобновить свое исполнение после приостановки и завершиться после вычисления Т-функции.

Очевидно, что такому набору свойств соответствует понятие потока управления (далее — просто потока), элемента концепции множественных потоков управления (англ. термин threads[1]), реализуемых, в частности, стандартной библиотекой Posix Threads. В то же время многие выполненные реализации модели вычислений «самотрансформация вычисляемой сети», и в частности, Т-система, используют собственные реализации базовой поддержки понятия Т-процесса, такие, как Tthreads [2]. Дело в том, что Т-процессы существенно отличаются от потоков тем, что диспетчеризируются не принудительно извне, в зависимости от функционирования системного таймера и алгоритмов диспетчера ядра ОС, а кооперативно (англ. cooperative scheduling), то есть по условиям, возникающим в процессе вычислений (например, засыпают по неготовности необходимого значения Т-переменной, возобновляют выполнение после присваивания вычисленного значения в Т-переменную Т-процессом-поставщиком).

Конечно, принципы кооперативной диспетчеризации могут быть реализованы и с использованием стандартных потоков, поддерживаемых ядром ОС, и системных объектов, таких, как семафоры, очереди, файловые дескрипторы и т.п., но следствием этого будет необходимость многократного перехода при переключении Т-процессов из пользовательского режима исполнения программы в системный (с последующим возвратом в пользовательский режим), что связано с довольно значительными накладными расходами в части времени выполнения и, как следствие, способно вызвать деградацию производительности.

Кроме того, для стека каждого из стандартных системных потоков резервируется значительная память (не менее 4Кб), что может в десятки раз превышать память, необходимую для стека Т-процесса, вычисляющего ту или иную Т-функцию, а это при значительном числе передававшихся на исполнение Т-процессов выразится в неэффективном использовании памяти. К тому же результату может привести и использование системных объектов для целей управления переключением Т-процессов.

Таким образом, реализация Т-процессов на основе кооперативной диспетчеризации в пользовательском режиме исполнения является

[1]В русскоязычной литературе используются также эквивалентные термины *нити управления* (или даже просто *нити*).

предпочтительным вариантом. Данная реализация может быть выполнена за счет использования механизма сопрограмм, поддержанного в алгоритмическом языке C стандартными библиотечными функциями setjmp/longjmp (обычно реализуются специальные формы таких функций, «облегченные» с целью отказа от использования системных вызовов, вызывающих нежелательные переключения контекстов ядра ОС и пользовательских процессов).

Облегченные потоки исполнения можно было бы назвать *суперлегковесными процессами* или *легковесными потоками* (в данной статье, при необходимости, будет использоваться второй из этих терминов).

Удобным и эффективным вариантом реализации Т-процессов на основе легковесных потоков является выбор фиксированного набора таких потоков в качестве потоков-диспетчеров (в соответствии с числом процессорных ядер), при этом каждый из потоков-диспетчеров выполняется внутри соответствующего ему системного потока ОС, а каждый из регулярных легковесных потоков является базой для реализации некоторого Т-процесса, выполняется на том же системном потоке, что и поток-диспетчер и «знает о существовании» только выбравшего его на исполнение потока-диспетчера, переключаясь именно на него в случае необходимости.

Такая реализация очень напоминает конструкцию, используемую в шаблоне программирования «команда рабочих» (англ. worker-crew [**3**, стр. 394]), реализованного на основе пула потоков (англ. thread pool [**4**]) с фиксированным числом потоков. Близкая по свойствам реализация пула потоков встречается в базовой библиотеке современной версии языка Java (инфраструктура Fork/Join [**5**]).

В то же время существенное отличие понятия легковесного потока от используемого в концепции пула потоков понятия задачи (англ. task) заключается в том, что для легковесного потока приостановка и продолжение исполнения являются регулярными базовыми действиями, основанными на сохранении и восстановлении состояния данного потока, в то время, как для задачи в пуле потоков такие действия являются либо чрезвычайными, приостанавливающими текущее выполнение потока ОС (и потенциально способными привести к созданию нового потока ОС в пуле и даже к полной блокировке всего процесса выполнения приложения), либо крайне ограниченными — например, включающими только примитив ожидания завершения потока-потомка (join) в инфраструктуре Fork-Join.

К сожалению, стандартные библиотеки языка Java не содержат сколь-либо близких аналогов функций setjmp/longjmp, поэтому изучение подходов к реализации понятия Т-процесса на базе легковесных потоков для платформы JVM оказывается самостоятельным (и небезынтересным) предметом исследования.

2. Реализация легковесных потоков для платформы JVM

2.1. Анализ возможных подходов к реализации легковесных потоков для платформы JVM

При анализе возможных подходов к реализации легковесных потоков для платформы JVM наиболее естественными кажутся следующие две возможности:

(1) реализовать поддержку базовых примитивов работы с легковесными потоками, образно выражаясь, в «ядре платформы JVM». В этом случае изменения вносятся в код самого интерпретатора JVM-байткода, и дополнительно может быть использована программная библиотека, включающая в свой состав низкоуровневый аппаратурно-зависимый (англ. “native”) код;
(2) использовать стандартные потоки в качестве реализации легковесных потоков, совместно со стандартными примитивы языка Java, предназначенными для управления потоками.

Первый из этих двух вариантов дает возможность в полном объеме реализовать легковесные потоки — за счет модификации (адаптации к нуждам реализации) алгоритмов и базовых внутренних структур данных JVM. Это обеспечивает условия для реализации управления памятью, выделяемой для стеков легковесных потоков, а также для реализации примитивов создания легковесных потоков, передачи их на исполнение, приостановки, возобновления и т.п. Еще одна серьезная положительная сторона данного варианта — потенциальная возможность обеспечить высокую эффективность такой реализации в части производительности (за счет максимального снижения дополнительных накладных расходов). И поскольку основная цель использования параллелизма — повышение суммарной производительности аппаратно-программной платформы, использование именно такого, ориентированного на максимальную эффективность, подхода к реализации легковесных потоков на первый взгляд представляется наиболее естественным.

Единственной, хотя и весьма существенной, отрицательной чертой такой реализации является ее чрезвычайно высокая трудоемкость. Изменения, которые были бы способными обеспечить достаточный уровень эффективности, могут, вероятно, затронуть значительную часть исходного кода «ядерной» части JVM, в том числе будет, скорее всего, затронут и код подсистемы, ответственной за осуществление «сборки мусора». В этих условиях обеспечить стабильность функционирования данной реализации было бы затруднительно, что, в частности, подтверждается результатами исследования [6], проведенного при поддержке фирмы Oracle: по утверждению автора [7], поддержать относительную стабильность функционирования удается только в случае упрощенной реализации, в которой, в частности, отсутствует возможность миграции легковесных потоков между обеспечивающими их исполнение стандартными потоками (что необходимо для полноценной балансировки загрузки процессорных ядер, в конечном счете — для обеспечения высокой производительности).

Второй вариант — отказ от использования легковесных потоков и замена их на стандартные потоки — позволяет избежать модификации кода JVM и таким образом добиться радикального снижения, по сравнению с альтернативным вариантом, трудоемкости реализации. При этом, однако, теряются преимущества, предоставляемые использованием легковесных потоков, что в конечном счете должно привести к снижению эффективности реализации.

2.2. Струи — новый подход к реализации легковесных потоков

В последние годы наблюдается существенный рост интереса к такой модели вычислений и парадигме, как акторы [8]. И это не случайность, поскольку использование акторов являются одним из подходов к снижению трудоемкости разработки параллельных приложений и повышению эффективности использования параллелизма аппаратуры. Естественно, что реализация этой парадигмы была выполнена и для платформы JVM (см., например, [9]).

Эффективная реализация акторов на платформе JVM требует, чтобы на данной платформе была в той или иной форме выполнена реализация легковесных потоков. Такие реализации появились [10] и в изолированной форме, пригодной для использования вне зависимости от реализации акторов. Для именования подобных реализаций

легковесных потоков (и не только для платформы JVM) стал использоваться термин *струи* (англ. fibers[2]).

Струи для JVM могут быть реализованы аналогично механизму задач/пула потоков (или даже просто на основе данного механизма). К реализации задач добавляется механизм формирования *продолжения* (англ. continuation) — сохранения состояния стека и аналога счетчика команд, т.е. той точки кода, где вычисление было приостановлено. При необходимости продолжить вычисление создается новая задача (старая уже завершена), которая по сохраненным внутри продолжения данным восстанавливает состояние вычисления, после чего струя продолжает вычисление с точки приостановки — вплоть до завершения или следующей приостановки. Таким образом, можно сказать, что струя сама по себе является реализацией парадигмы продолжение.

Сохранение состояния вычисления реализуется с использованием инструментации сгенерированного байткода. Под инструментацией понимается внесение изменений в находящийся в класс-файле байткод; такая модификация может быть выполнена как предварительно (англ. *AOT* — "ahead of time"), так и непосредственно при загрузке класс-файла в процессе работы приложения — в последнем случае используется механизм Java-агента[3]. В процессе инструментации производится анализ методов класса, с целью выяснить, существует ли возможность приостановки данного метода в процессе вычислений — (включая приостановку в методах, вызванных из данного метода). Лишь те методы, которые потенциально могут быть приостановлены, действительно подвергаются изменениям.

Основная идея трансформации иллюстрируется[4] на рис. 1. Рассматривается случай, когда приостановка осуществляется за счет генерации исключения. В случае, если приостановка реализуется с использованием обычного возврата из метода, суть преобразований не меняется, хотя конечно, некоторые частности могут различаться.

---

[2]волокна. Как волокна являются составными частями нити (англ. thread), так и струи входят в состав потоков.

[3]Для этого при запуске виртуальной машины используется дополнительный аргумент -javaagent. Описание механизма Java-агента доступно в составе интернет-документации фирмы Oracle по пакету java.lang.instrument по URL: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

[4]Поскольку преобразование выполняется над байткодом JVM, пример приводится на Java-подобном псевдокоде, в частности, из-за присутствия инструкций goto.

```java
@Suspendable
f(int n) {
  int i = 0;
  do {
    g();
  } while (++i < n/2);
  ...
}

@Suspendable
g() {
  ...
}
```

```java
@Suspendable
f(int n) {
  int i;
  try {                          // Прелюдия
    Fiber fiber = Fiber.getCurrentFiber();
    if (fiber.resuming)
      switch(fiber.pc(f)) {
      case G0: goto G0;
      ...
      }
    i = 0;
    do {
      fiber.pushmethod (f, G0);
      fiber.save(i);
      G0:                          // pre-call
      i = fiber.restore(0);
      g();
    fiber.free(0);                 // post-call
    } while (++i < n/2)
     ...
  catch(SuspendExecution e) {
    throw e;
  }
}

@Suspendable
g() {
   ...
}
```

Рис. 1. Пример инструментации метода (слева – исходный метод, справа – инструментированный)

Каждый приостанавливаемый метод f инструментируется следующим образом: он сканируется для поиска вызовов других приостанавливаемых методов. Для каждого вызова приостанавливаемого метода g, до вызова g вставляется некоторый код (строки 11-15), который сохраняет (и восстанавливает) состояние локальных переменных в стек струи (струя — как структура данных — включает отдельный стек), и фиксирует тот факт, что данный вызов g — возможная точка приостановки.

Если в процессе исполнения g на самом деле заблокируется, будет сгенерировано исключение (обозначим его тип как SuspendExecution),

которое, в конечном счете будет перехвачено одним из начальных базовых методов струи. Исполнение соответствующей задачи в пуле потоков данного приложения будет завершено, но состояние струи на момент приостановки будет сохранено в структуре Fiber. В момент, когда будет выполняться возобновление функционирования струи, в пуле потоков будет запущена новая задача, которая начнет восстанавливать сохраненное состояние струи.

Метод f будет вызван, и протокол исполнения (сформированный в процессе исполнения струи отдельный стек и дополнительные переменные) покажет, что мы были заблокированы при обращении к g, так что мы немедленно перейдем к строке в f, где восстанавливается состояние локальных переменных, а затем вызывается g, и выполним вызов. В конце концов мы доберемся до действительной точки засыпания, где мы продолжим выполнение с точки, непосредственно следующей за вызовом. Когда управление возвратится в f, код, внесенный вслед за вызовом g, освободит часть стека струи, которая хранила восстановленные локальные переменные f.

3. Поддержка понятия Т-процесса в реализации экспериментального языка программирования ajl

В качестве основы для поддержки Т-процессов в реализуемом экспериментальном языке программирования ajl была использована содержащая реализацию струй свободно-доступная[5] библиотека Quasar с открытым исходным кодом. Quasar — динамично развивающаяся библиотека (на момент написания публикации был доступен выпуск версии 0.7.3) с хорошей поддержкой и обратной связью с пользователями.

Важной для воплощаемой языком ajl модели вычислений «самотрансформация вычисляемой сети» особенностью данной библиотеки является поддержка совместимых с реализацией струй переменных, способных содержать неготовые значения. Реализация Т-процессов на основе струй содержит реализацию следующих основных примитивов:

spark — создание и запуск нового Т-процесса. Часть работы по исполнению данного примитива составляет формирование набора аргументов Т-функции (являющихся Т-переменными), а также установление соответствия между набором результатов Т-функции и теми Т-переменными, в которые возврат этих результатов должен быть осуществлен;

[5]URL: http://docs.paralleluniverse.co/quasar/

send — возврат одного из результатов вызова T-функции. Может привести к переходу той T-переменной, в которую возвращается результат, из неготового состояния в готовое. Отметим, что в качестве результата может быть возвращено и неготовое значение — результат другого вызова;

getvalue — получение значения, содержащегося в некоторой переменной, включая возможное ожидание готовности значения, содержащегося в данной T-переменной.

Как видим, все т ри базовых примитива языка ajl содержат операции с T-переменными, способными содержать неготовые значения. Таким образом, наличие в библиотеке Quasar готовой поддержки манипуляции неготовыми значениями является подспорьем при реализации языка. Отметим, что Quasar поддерживает операции с неготовыми значениями не только для струй, но и для потоков. С этой целью в Quasar было введено понятие *течения* (англ. strand[6]), которое является обобщением понятий потока и струи.

4. Экспериментальное сравнение реализации понятия T-процесса на основе потоков и струй

Для сравнения потенциала разрабатываемой реализации T-процессов в языке ajl на основе струй и аналогичной реализации на основе потоков было разработано тестовое параллельное приложение, ориентированное на использование возможностей многоядерных процессоров. Приложение разрабатывалось на языке Java с использованием библиотеки Quasar. Суть алгоритма распараллеливания, использованного в задаче, заключается в рекурсивном делении общего объема работы на независимые фрагменты; для исполнения каждого из таких фрагментов выполняется запуск отдельного течения — струи или потока. Это вновь запущенное течение, имитируя «рабочий» T-процесс, выполняет заданное число циклов, наращивающих на 1 значение локальной переменной-счетчика (далее в тексте данной статьи будем для краткости называть такое течение «работником»). На каждом уровне рекурсии количество циклов делилось пополам между двумя ветвями, каждая из ветвей запускалась как отдельное течение. Таким образом, от глубины рекурсии зависело как число имитируемых T-процессов, так и количество циклов, выполняемых каждым из «работников»; при этом общее, суммарное число циклов,

[6] обобщение понятий fiber и thread.

Таблица 1. Результаты экспериментального сравнения различных реализаций понятия T-процесса

кол-во «работников»	Время исполнения приложения в секундах		
	в статике, AOT	динамическая инструментация	имитация потоками
1	55.60	56.06	55.98
2	29.35	29.54	29.57
4	14.72	15.10	15.05
8	14.74	15.08	15.03
16	14.75	15.07	15.02
32	14.74	15.08	15.00
64	14.76	15.06	15.02
128	14.75	15.08	15.03
256	14.76	15.09	15.05
512	14.77	15.09	15.09
1024	14.78	15.10	15.18
2^{11}	14.79	15.12	15.68
2^{12}	14.83	15.18	16.90
2^{13}	14.91	15.25	19.35
2^{14}	14.98	15.23	26.88
2^{15}	15.03	15.32	52.24
2^{16}	15.16	15.44	166.58
2^{17}	15.53	16.08	
2^{18}	16.38	16.43	
2^{19}	17.45	17.97	
2^{20}	20.84	21.52	
2^{21}	31.09	32.35	

исполнившихся в рамках всего приложения, оставалось постоянным (в эксперименте оно равнялось $2^{33} \approx 10^{10}$). В процессе проведения эксперимента измерялось общее время выполнения приложения для различного числа «работников».

Эксперимент проводился на ЭВМ, оснащенной ОЗУ объемом 16 Гбайт и 4-ядерным процессором Intel® Core™i7 930, работающим на частоте 2.80 ГГц. ЭВМ в ходе эксперимента функционировала под управлением 64-битной ОС Linux с ядром версии 3.16.7. Для исполнения приложения использовалась 64-битная виртуальная машина языка Java, использующая JIT-компилятор HotSpot из состава дистрибутива OpenJDK версии 7u79.

Результаты эксперимента приведены в таблице 1 и на графике (см. рис. 2).

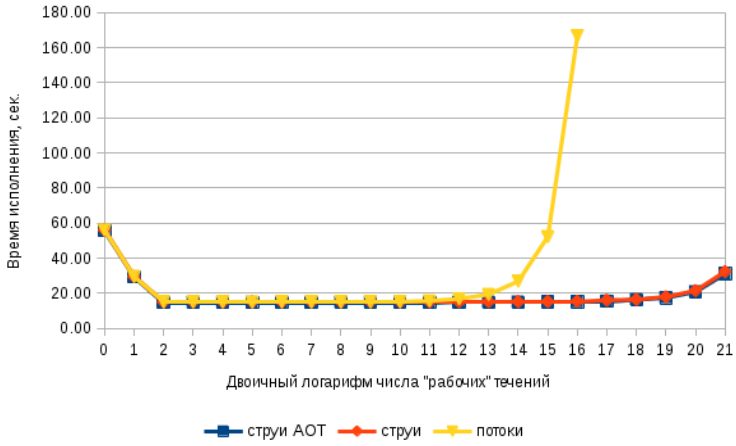

Рис. 2. Экспериментальный график зависимости времени исполнения от логарифма числа «работников»

Из приведенных в таблице 1 данных видно, что вариант реализации приложения, основанный на использовании струй и предварительной инструментации, демонстрирует наименьшие времена исполнения во всем диапазоне значений числа «работников».

При низких числах «работников» — в диапазоне от 1 до 1024 включительно — времена исполнения различных вариантов приложения для каждого фиксированного значения числа «работников» отличаются не более, чем на десятые доли процента. При изменении числа работников от 1 до 4 время исполнения приложения падает практически линейно, что соответствует распределению вычислительной нагрузки («работников») между ядрами процессора; затем — при нарастании вплоть до 1024 — времена исполнения приложения составляют около 15 секунд и изменяются незначительно, поскольку число циклов, приходящееся на каждое процессорное ядро, остается практически постоянным, а накладные расходы на создание и планирование такого количества струй и потоков не слишком велики.

В диапазоне чисел «работников» от 2^{11} до 2^{16} времена исполнения варианта, основанного на потоках, заметно нарастают (а уровни производительности и эффективности использования вычислительной мощности, соответственно, падают) по сравнению с вариантами реализации приложения, основанных на использовании струй, что

можно объяснить нарастанием накладных расходов в ядре ОС на создание и управление такими значительными числами потоков. Для числа «работников», равного 2^{16}, время исполнения варианта приложения, использующего потоки, уже более чем в 10 раз превышает значение времени исполнения для вариантов, использующих струи. Для чисел «работников», больших 2^{16}, нормальное функционирование варианта реализации приложения, основанного на использовании потоков, становится невозможным из-за возникающего при исполнении приложения отказа пары JVM/ядро ОС создавать новые потоки.

Оба варианта приложения, основанные на использовании струй для реализации «работников», демонстрируют относительную стабильность времени выполнения вплоть до числа «работников», равного 2^{17}. Начиная с этого количества «работников», времена исполнения приложения начинают нарастать, но менее сильно, чем это было в случае реализации, основанной на потоках (подчеркнем тот факт, что вариант, основанный на потоках, при таких значениях числа «работников» функционировать уже не способен). Для числа «работников», равного 2^{21}, время исполнения для обоих вариантов составляет чуть более 30 секунд, что только в 2 раза превышает времена исполнения для числа «работников», равного 2^{10}. Можно видеть, что использование струй обеспечивает более-менее приемлемое функционирование приложения для чисел «работников», примерно в 100 раз больших, чем в случае использования потоков. В момент, когда число «работников» достигает 2^{22}, нормальное функционирование вариантов реализации приложения, основанных на использовании струй, не обеспечивается из-за невозможности выделения запрашиваемой приложением памяти.

Таким образом, на основе полученных экспериментальных данных можно сделать вывод, что при реализации языка ajl использование струй позволит обеспечить для значительных чисел вызовов Т-функций выигрыш в производительности и эффективности. Кроме того, это обеспечит возможность нормального функционирования разрабатываемых с использованием данного языка приложений при гораздо более высоких значениях числа активных Т-процессов.

Заключение

В данной публикации рассмотрены различные варианты подходов к реализации понятия «Т-процесс» параллельной модели вычислений «самотрансформация вычисляемой сети» для платформы JVM — виртуальной машины языка Java.

С учетом наличия потенциальных проблем, связанных с реализацией данного понятия как на основе классических потоков ОС/JDK, так и в случае внесения поддержки легковесных потоков непосредственно в код виртуальной машины, предложен подход к реализации Т-процессов, основанный на использовании понятия струй, т.е. легковесных потоков, реализуемых вне ядра JVM.

Приведены результаты экспериментального сравнения подходов к реализации понятия «Т-процесс», основанных на использовании классических потоков и струй.

Показано, что использование струй для реализации модели вычислений «самотрансформация вычисляемой сети» при значительном числе активных Т-процессов позволяет как повысить эффективность использования вычислительной мощности ЭВМ, так и обеспечить более устойчивое функционирование приложения, выполненного с использованием реализации данной модели вычислений, используемой в разрабатываемом языке параллельного программирования ajl.

Список литературы

[1] S. M. Abramov, A. I. Adamowitch, I. A. Nesterov, S. P. Pimenov, Y. V. Shevchuck, “Multiprocessor with automatic dynamic parallelizing”, *NATUG 6-th Meeting ”Transputer: Research and Application”*, IOS Press, Vancouver, 1993, pp. 333–344. ↑178

[2] С. М. Абрамов, А. И. Адамович, М. Р. Коваленко. «Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки лучей», *Программирование*, **25**:2 (1999), с. 100–107. ↑178,181

[3] R. P. Garg, I. Sharapov. *Techniques for Optimizing Applications: High Performance Computing*, Prentice-Hall, 2002, 616 p. ↑182

[4] L. Yibei, T. Mullen, X. Lin. “Analysis of optimal thread pool size”, *ACM SIGOPS Operating Systems Review*, **34**:2 (2000), pp. 42–55. ↑182

[5] Г. Шилдт. *Java. Полное руководство*, ООО "И.Д. Вильямс", М., 2012, 1104 с. ↑182

[6] L. Stadler, T. Wurthinger, C. Wimmer, “Efficient coroutines for the Java platform”, *8th International Conference on the Principles and Practice of Programming in Java*, ACM, New York, 2010, pp. 10–19. ↑184

[7] L. Stadler. *Coroutines for Java*, URL: http://ssw.jku.at/General/Staff/LS/coro/ ↑184

[8] C. Hewitt, P. Bishop, R. Steiger, “A universal modular ACTOR formalism for artificial intelligence”, *3rd International Joint Conference on Artificial Intelligence IJCAI'73*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245. ↑184

[9] P. Haller, M. Odersky, “Event-based programming without inversion of control”, *Modular Programming Languages*, 7th Joint Modular Languages Conference, JMLC 2006 (Oxford, UK, September 13–15, 2006), Lecture Notes in Computer Science, vol. **4228**, Springer, Berlin–Heidelberg, 2006, pp. 4–22. ↑184

[10] S. Srinivasan, A. Mycroft, “Kilim: Isolation-Typed Actors for Java”, *22nd European Conference on Object-Oriented Programming ECOOP'08*, Springer-Verlag, 2008, pp. 104–128. ↑184

Рекомендовал к публикации *к.ф.-м.н. А. В. Климов*

Об авторе:

Алексей Игоревич Адамович

Старший научный сотрудник ИПС им. А.К. Айламазяна РАН. Работает в области инструментальных средств для разработки параллельных приложений. Разработчик первой параллельной версии Т-системы, ведущий разработчик параллельного отладчика tdb. Активный участник суперкомпьютерного проекта «СКИФ» Союзного государства России и Беларуси.

e-mail: lexa@adam.botik.ru

Пример ссылки на эту публикацию:

А. И. Адамович. «Струи как основа реализации понятия Т-процесса для платформы JVM», *Программные системы: теория и приложения*, 2015, **6**:4(27), с. 177–195.

URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf

Alexei Adamovich. *Fibers as the basis for the implementation of the notion of the T-process for the JVM platform.*

Abstract. The spread and the accessibility of modern parallel computing platforms shows the lack of an appropriate support level of the needs of software developers for the parallel applications implementation. At PSI RAS the approach is under development to parallelization of programs based on the use of the computation model named "self-transformation of computational network". This paper discusses the various options for approaches to the implementation of the notion "T-process" — one of the basic notions of the computation model — for the JVM platform. The potential problems are analyzed associated with the implementation of the " T-process" notion on the basis of the classical OS/JDK threads, and in the event of a lightweight threads are supported directly in the virtual machine code. The approach to the implementation of the T-processes is proposed based on the use of the concept of the fibers, i.e. lightweight threads, which are implemented outside the JVM code. The results of experimental comparison of different approaches to the implementation of the notion of "T-process" based on the use of fibers and classical streams are provided. We also analyze the effect of the use of fibers in the implementation of the "self-transformation of the computational network" computation model which is used in the ajl parallel programming language being developed for the JVM platform.(*in Russian*).

Key Words and Phrases: programming languages implementation, parallel computations, JVM platform, automatic dynamic parallelization, threads, fibers.

References

[1] S. M. Abramov, A. I. Adamowitch, I. A. Nesterov, S. P. Pimenov, Y. V. Shevchuck, "Multiprocessor with automatic dynamic parallelizing", *NATUG 6-th Meeting "Transputer: Research and Application"*, IOS Press, Vancouver, 1993, pp. 333–344.

[2] S. M. Abramov, A. I. Adamovich, M. R. Kovalenko. "T-sistem — a programming environment with a support for automatic dynamic parallelizing of programs. An example of the algorithm for constructing images using ray tracing", *Programmirovaniye*, **25**:2 (1999), pp. 100–107.

[3] R. P. Garg, I. Sharapov. *Techniques for Optimizing Applications: High Performance Computing*, Prentice-Hall, 2002, 616 p.

[4] L. Yibei, T. Mullen, X. Lin. "Analysis of optimal thread pool size", *ACM SIGOPS Operating Systems Review*, **34**:2 (2000), pp. 42–55.

[5] H. Shildt. *Java. The Complete Reference*, OOO "I.D. Vil'yams", M., 2012, 1104 p.

[6] L. Stadler, T. Wurthinger, C. Wimmer, "Efficient coroutines for the Java platform", *8th International Conference on the Principles and Practice of Programming in Java*, ACM, New York, 2010, pp. 10–19.

[7] L. Stadler. *Coroutines for Java*, URL: http://ssw.jku.at/General/Staff/LS/coro/

[8] C. Hewitt, P. Bishop, R. Steiger, "A universal modular ACTOR formalism for artificial intelligence", *3rd International Joint Conference on Artificial Intelligence IJCAI'73*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.

[9] P. Haller, M. Odersky, “Event-based programming without inversion of control”, *Modular Programming Languages*, 7th Joint Modular Languages Conference, JMLC 2006 (Oxford, UK, September 13–15, 2006), Lecture Notes in Computer Science, vol. **4228**, Springer, Berlin–Heidelberg, 2006, pp. 4–22.

[10] S. Srinivasan, A. Mycroft, “Kilim: Isolation-Typed Actors for Java”, *22nd European Conference on Object-Oriented Programming ECOOP’08*, Springer-Verlag, 2008, pp. 104–128.

Sample citation of this publication:

Alexei Adamovich. “Fibers as the basis for the implementation of the notion of the T-process for the JVM platform”, *Program systems: theory and applications*, 2015, **6**:4(27), pp. 177–195. (*In Russian*).

URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf