APPLICATION PROGRAMMING INTERFACE FOR FUNCTIONAL PROGRAMMING FOR PARALLEL AND DISTRIBUTED SYSTEMS

I. Petriakov, I. Gankevich

Saint Petersburg State University, 13B Universitetskaya Emb., Saint Petersburg, 199034, Russia

E-mail: i.gankevich@spbu.ru

There are a huge amount of scientific and commercial applications written with a focus on sequential execution. Running such programs on multiprocessor systems is possible, but without taking advantage of these systems. To execute a program with these capabilities in mind, it is often necessary to rewrite the program. However, this is not always the best choice. In this work, the possibility of parallel execution of programs written in functional languages is considered, the principle of operation of the proposed interpreter of a functional programming language is described in detail. As an example of functional language was chosen Guile. Parallelism in it is achieved through parallel execution of function arguments. The result of this work can be used as an example of building programming interfaces for other programming languages.

Keywords: distributed computing, parallel computing, Guile.

Ivan Petriakov, Ivan Gankevich

Copyright © 2021 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3041/606-611-paper-112.pdf

ПРОГРАММНЫЙ ИНТЕРФЕЙС ДЛЯ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ ДЛЯ ПАРАЛЛЕЛЬНЫХ И РАСПРЕДЕЛЕННЫХ СИСТЕМ

И. Петряков, И. Ганкевич

Санкт-Петербургский государственный университет,
Россия, 199034, Санкт-Петербург, Университетская наб., д. 7–9

E-mail: i.gankevich@spbu.ru

Существует огромное количество научных и коммерческих приложений, написанных с прицелом на последовательное исполнение. Запуск таких программ на многопроцессорных системах возможен, но без использования преимуществ этих систем. Для выполнения программы с учетом этих возможностей зачастую необходимо переписать программу. Однако, это не всегда оптимальный выбор. В этой работе рассматривается возможность параллельного выполнения программ, написанных на функциональных языках, подробно описывается принцип работы предложенного интерпретатора функционального языка программирования. В качестве примера была выбрана реализация языка Scheme – Guile. Параллелизм в нем достигается за счет параллельного выполнения аргументов функции. Результат данной работы может быть использован как пример построения программных интерфейсов для других языков программирования.

Ключевые слова: распределенные вычисления, параллельные вычисления, Guile.

Петряков Иван, Ганкевич Иван

Copyright © 2021 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

1. Введение

Один из способов создания программного интерфейса высокого уровня для суперкомпьютеров и кластеров — изменить существующий функциональный язык для параллельного выполнения программы на кластере. Преимущество такого подхода заключается в том, что если у вас уже есть последовательная программа, написанная на функциональном языке, то вы можете выполнить ее на кластере с использованием либо компилятора, который генерирует параллельный код, либо библиотеки, которая предоставляет те же функциональные формы (например, *map*, *reduce*), которые реализованы для параллельного выполнения на нескольких узлах кластера. Однако, этот подход представляет некоторые сложности: обработка спекулятивного выполнения различных ветвей кода и устойчивость к сбоям узлов кластера. Вероятно, главный недостаток этого подхода заключается в том, что на функциональных языках написано не так много высокопроизводительных приложений: большинство из них из соображений эффективности написано на низкоуровневых императивных языках, которые не предоставляют средств выполнения на кластере.

2. Архитектура

Для создания низкоуровневого языка для параллельных и распределенных вычислений был использован фреймворк Subordination [1]. Он заимствует знакомый из последовательных низкоуровневых языков стек вызовов функций и дополняет его асинхронными вызовами функций и возможностью чтения и записи кадров стека вызовов.

Чтобы понять принцип работы этого фреймворка, нужно иметь представление о работе стека вызовов в привычном его понимании. Мы представляем каждый кадр стека объектом: локальные переменные становятся полями объекта, и каждый вызов функции разбивается на код, который предшествует вызову функции, тело функции, и код, следующий за ним. Код, который идет до вызова, помещается в метод «act» объекта, а для выполнения этого кода создается новый объект для асинхронного вызова функции. Код, который идет после вызова, помещается в метод «react» объекта, и этот код вызывается асинхронно при возврате из функции (этот метод принимает соответствующий объект в качестве аргумента). У объекта также есть методы «read» и «write», которые используются для чтения и записи его полей в файл и из файла или для копирования объекта на другой узел кластера. В этой модели каждый объект содержит достаточно информации для выполнения соответствующего вызова функции, и мы можем выполнять эти вызовы в любом порядке. Кроме того, объект является самодостаточным, и мы можем попросить другой узел кластера выполнить вызов функции или сохранить объект на диск для отложенного выполнения вызова, когда пользователь хочет возобновить вычисления (например, после обновления и перезагрузки компьютера).

Эти объекты являются отличительной особенностью Subordination по сравнению с другими инструментами параллельного программирования. Возможность указывать иерархические зависимости между объектами, обрабатываемыми параллельно, делает очевидным механизм устойчивости к сбоям. Если вызов какого-либо метода объекта завершается неудачей, то иерархические связи легко позволяют определить родительский объект, который становится ответственным за повторное выполнение метода подчиненного объекта, завершившегося неудачей, на оставшихся узлах кластера. Чтобы повторно выполнить метод объекта, у которого нет родительского, создается его копия и отправляется на другой узел.

Управляющие объекты реализуют логику потока управления в своих методах «act» и «react» и хранят состояние текущей ветви потока управления. И логика, и состояние реализуются программистом. В методе «act» некоторая функция либо вычисляется напрямую, либо разбивается на вызовы вложенных функций (представленных набором подчиненных объектов), которые впоследствии отправляются в очередь на выполнение. В методе «react» подчиненные объекты, возвращенные из конвейера, обрабатываются их родителем. Вызовы методов «act» и «react» являются асинхронными и выполняются в потоках, связанных с

очередью. Для каждого объекта «act» вызывается только один раз, а для нескольких объектов вызовы выполняются параллельно друг другу, тогда как метод «react» вызывается один раз для каждого подчиненного объекта, и все вызовы выполняются в одном потоке для предотвращения состояний гонки.

3. Ручной параллелизм

Подход к написанию параллельных программ с использованием ядер — это не что иное, как старая модель стека вызовов функций, но с асинхронным выполнением каждого вложенного вызова функций и возможностью скопировать стек на другой узел кластера для выполнения функции там. Следовательно, его можно использовать как модель, в которую транслируются существующие функциональные языки программирования. Самый простой способ перевести язык программирования подобный LISP в эту модель — выполнить параллельное вычисление всех аргументов процедуры, т. е. транслировать каждый вызов процедуры в программе в родительский объект, который создает подчиненный объект для каждого аргумента процедуры, чтобы вычислить его параллельно с другими подчиненными объектами и возвратить результат в родительский объект. Родительский объект, в свою очередь, выполняет процедуру как обычно с уже вычисленными аргументами. Хотя этот подход прост, процедуры LISP часто работают с одним аргументом, который является списком. Чтобы реализовать эти процедуры с использованием управляющих объектов, мы должны перевести языковые формы, такие как *map* и *fold* (на которых основано большинство таких процедур), в управляющие объекты, а затем реализовать процедуры с использованием этих форм.

4. Автоматический параллелизм

Отличительной чертой языков подобных LISP является гомоиконичность, то есть представление кода и данных древовидной структурой (в виде списков, которые могут содержать другие списки в качестве элементов). Эта особенность позволяет выразить параллелизм с помощью списков: каждый элемент списка может быть вычислен независимо и может быть отправлен на другие узлы кластера для параллельных вычислений. Для реализации параллелизма был создан интерпретатор языка Guile [2], который вычисляет каждый элемент списка параллельно с использованием управлящих объектов. На практике это означает, что каждый аргумент вызова процедуры (вызов процедуры также является списком, первым элементом которого является имя процедуры) вычисляется параллельно (см. рис. 1). Этот интерпретатор может запускать любую существующую программу Guile (при условии, что она не использует явно параллельные потоки, блокировки и семафоры), и вывод будет таким же, как и в исходном интерпретаторе, программа будет автоматически использовать узлы кластера для параллельных вычислений, а отказоустойчивость будет автоматически предоставлена кластерным планировщиком.

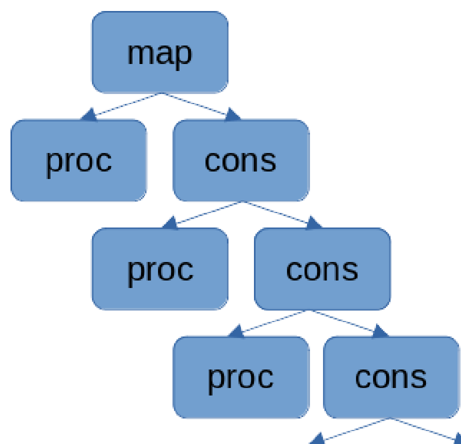

Рисунок 1. Схема выполнения процедуры map на интерпретаторе. Каждый прямоугольник выполняется в отдельном потоке.

Такой подход является наиболее прозрачным и безопасным способом написания параллельных и распределенных программ с четким разделением зон ответственности: программист заботится о логике приложения, а планировщик кластера заботится о параллелизме, балансировке нагрузки и отказоустойчивости. Интерпретатор состоит из стандартного цикла read-eval-print, из которого только этап eval использует управляющие объекты для параллельных и распределенных вычислений. Внутри *eval* используется гибридный подход для параллелизма: ядра используются для асинхронного вычисления аргументов вызовов процедур и аргументов примитива *cons*, только если эти аргументы содержат вызовы других процедур. Это означает, что все простые аргументы (переменные, символы, другие примитивы и т. д.) вычисляются последовательно без создания дочерних управляющих объектов.

5. Результаты

Чтобы протестировать введенные концепции, было разработано приложение, которое производит обработку данных NDBC, собираемых метеобуями по всему миру, и восстанавливает частотно-направленные волновые спектры из этих данных. Для этого нужно считать данные, собранные за 10 лет, обработать их и произвести необходимые вычисления. Как показано на рис. 2, предложенный подход параллелизма справился не хуже встроенных параллельных методов языка.Кроме того, предложенный подход позволяет запустить задачу на кластере. С увеличением узлов, разница между C++ версией и Guile сокращаются (см. рис. 3), что говорит об уменьшении накладных расходов. Однако, представленный интерпретатор языка показывает слишком большие накладные расходы в сравнении с остальными реализациями. Для интерпретатора был разработан синтетический тест, в котором функция спит 2 секунды для каждого элемента списка, имитируя тяжелую работу. В таком случае, когда затраты на выполнение программы больше накладных расходов, мы получаем прирост производительности (см. рис. 4).

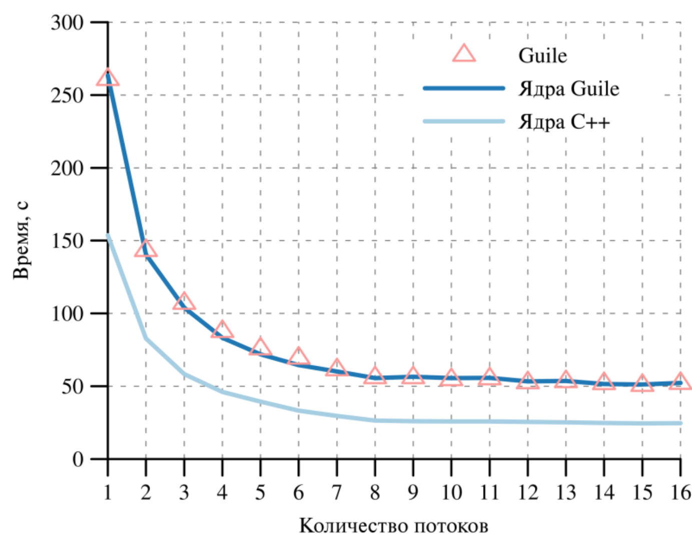

Рисунок 2. Зависимость времени выполнения задачи от количества потоков

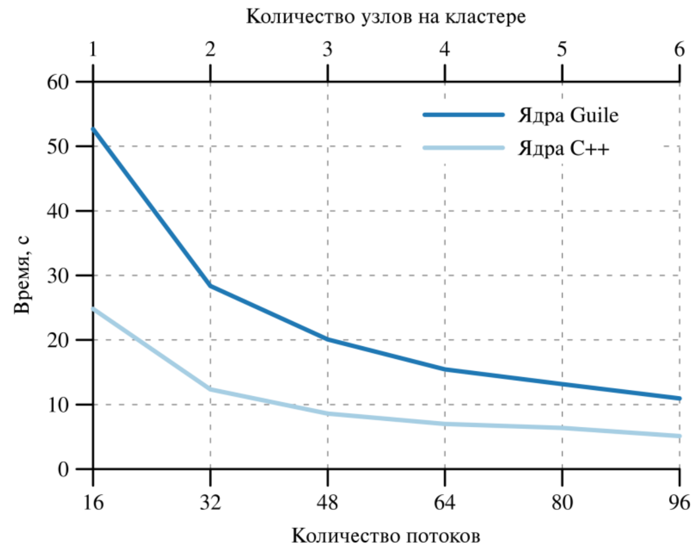

Рисунок 3. Зависимость времени выполнения синтетической задачи от количества потоков

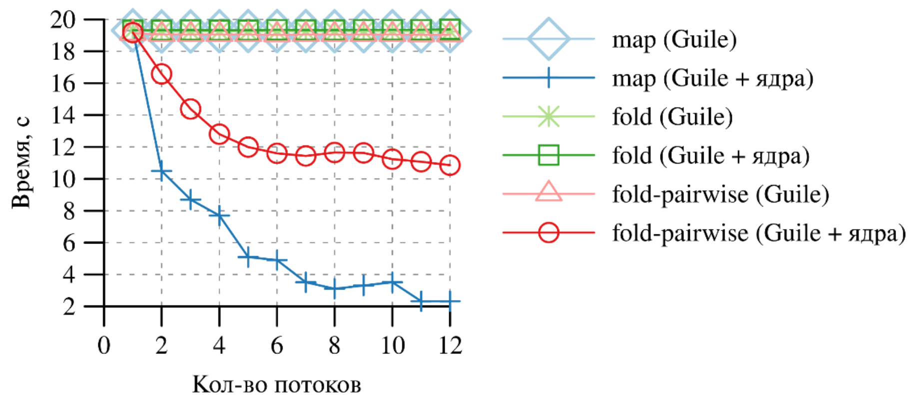

Рисунок 4. Зависимость времени выполнения синтетической задачи от количества потоков.

6. Заключение

Данную работу можно рассматривать как пример создания интерфейсов высокоуровнего языка программирования для выполнения вычислений на многопроцессорной системе. Результаты этой работы теперь являются частью фреймворка Subordination и распространяются под свободной лицензией GNU GPLv3.

Дальнешие работы будут направлены на уменьшение накладных расходов и распространение данной технологии на другие языки программирования.

Благодарности

Работа выполнена при поддержке Совета по грантам Президента Российской Федерации (грант № МК-383.2020.9).

Литература

[1] Gankevich I., Tipikin Y., Gaiduchok V. Subordination: Cluster management without distributed consensus //2015 International Conference on High Performance Computing & Simulation (HPCS). – IEEE, 2015. – С. 639-642.

[2] Galassi M. et al. Guile Reference Manual. – 2002.