Support for Parallel Computing in Julia Language

Dmitry S. Kulyabov[§¶], Migran N. Gevorkyan[§], Anna V. Korolkova[§], Leonid A. Sevastianov[§‖]

[§] *Department of Applied Probability and Informatics*
Peoples' Friendship University of Russia (RUDN University)
6 Miklukho-Maklaya St., Moscow, 117198, Russian Federation
[¶] *Laboratory of Information Technologies*
Joint Institute for Nuclear Research
6 Joliot-Curie, Dubna, Moscow region, 141980, Russian Federation
[‖] *Bogoliubov Laboratory of Theoretical Physics*
Joint Institute for Nuclear Research
6 Joliot-Curie, Dubna, Moscow region, 141980, Russian Federation

Email: kulyabov_ds@rudn.university, gevorkyan_mn@rudn.university, korolkova_av@rudn.university, sevastianov_la@rudn.university

The purpose of this paper is a brief review of tools for parallel computing implemented in the current version of the Julia Language.

Julia is a young promising language designed for scientific programming. Before describing directly the parallel programming capabilities of Julia we give a small overview of the main features of the language. We describe the main goals pursued by the authors of the language when it was created, and the ideology that they espoused. Separately discussed the scientific focus of Julia, features that make the language convenient for the needs of mathematical modelling, handling big data and intensive numerical computing. Separately discussed such a feature as Julia's multiple dispatch. This mechanism of language paying a lot of attention. Most built-in functions and operators using multiple dispatch.

In the second part of the paper we turn to the description of parallel programming. Julia is under active development, so support for parallel computing will be expanded, and the existing mechanisms may change. However, now Julia provides enough capabilities for writing quite complex programs using parallel computing. The basis for concurrency are processes (parallelism based on threads is not yet available). We describe the basic functions and macros that allow you to parallelize the execution of the functions, cycles and separate blocks of code. As the basis for presentation used the official guide and also the experience obtained by the authors in the process of using language.

The work is partially supported by RFBR grants No's 15-07-08795 and 16-07-00556. Also the publication was financially supported by the Ministry of Education and Science of the Russian Federation (the Agreement No 02.A03.21.0008).

Key words and phrases: scientific programming, parallel computing, Julia Language.

Copyright © 2017 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.
In: K. E. Samouilov, L. A. Sevastianov, D. S. Kulyabov (eds.): Selected Papers of the VII Conference "Information and Telecommunication Technologies and Mathematical Modeling of High-Tech Systems", Moscow, Russia, 24-Apr-2017, published at http://ceur-ws.org

О поддержке параллельных вычислений в языке Julia

Д. С. Кулябов[§¶], М. Н. Геворкян[§], А. В. Королькова[§], Л. А. Севастьянов[§‖]

[§] *Кафедра прикладной информатики и теории вероятностей, Российский университет дружбы народов, ул. Миклухо-Маклая, д. 6, Москва, Россия, 117198*

[¶] *Лаборатория информационных технологий, Объединённый институт ядерных исследований, ул. Жолио-Кюри, д. 6, г. Дубна, Московская область, Россия, 141980*

[‖] *Лаборатория теоретической физики, Объединённый институт ядерных исследований, ул. Жолио-Кюри, д. 6, г. Дубна, Московская область, Россия, 141980*

Email: kulyabov_ds@rudn.university, gevorkyan_mn@rudn.university, korolkova_av@rudn.university, sevastianov_la@rudn.university

Целью данной работы является краткий обзор средств для параллельных вычислений, реализованных в текущей версии Julia Language.

Julia — это молодой перспективный язык, предназначенный для научного программирования. Перед описанием непосредственно средств параллельного программирования мы даем небольшой обзор основных особенностей языка. Описываются основные цели, которые преследовали авторы языка при его создании, а также идеология, которой они придерживались. Отдельно обсуждается научная нацеленность Julia, особенности, которые делают данный язык удобным для нужд математического моделирования, обработки больших массивов данных и ресурсоемких вычислений. Отдельно обсуждается такая особенность Julia как множественная диспетчеризация. Данному механизму языка создатели Julia уделяют большое внимание и утверждают, что он в рамках Julia может заменить объектно-ориентированный подход. Большинство встроенных функций и операторов используют множественную диспетчеризацию.

Во второй части работы мы переходим к описанию средств параллельного программирования. Julia находится в стадии активной разработки, поэтому поддержка параллельных вычислений будет расширятся, а существующие механизмы возможно изменятся. Однако уже сейчас Julia обеспечивает достаточно средств для написания довольно таки сложных программ, использующих параллельные вычисления. Основой для параллелизма служат процессы (параллельность на основе потоков пока не доступна). Мы описываем основные функции и макросы, которые позволяют распараллелить выполнение функций, циклов и отдельных блоков кода. В качестве основы для изложения используется официальное руководство, а также опыт полученный авторами в процессе использования языка.

Работа частично поддержана грантами РФФИ № 15-07-08795, 16-07-00556. Также публикация подготовлена при финансовой поддержке Минобрнауки России (соглашение № 02.A03.21.0008).

Ключевые слова: научное программирование, параллельные вычисления, Julia.

1. Введение

Язык программирования Julia [1, 2] является динамическим, номинативным языком высокого уровня с параметрическим полиморфизмом. Язык разрабатывается в первую очередь для высокопроизводительных научных вычислений, хотя может использоваться и как язык общего назначения. Первая публичная версия языка была представлена в 2012 году, на начало 2017 года актуальной является версия 0.5. Несмотря на новизну, уже сейчас выходят книги с описанием языка и популярных библиотек, в том числе переведенные на русский язык [3].

Данная работа носит обзорный характер. Сначала дается общее описание основных особенностей языка, а затем более подробно описываются возможности параллельного программирования, поддерживаемые текущей версией языка.

2. Ключевые особенности языка

Так как Julia является динамическим языком, то в нем тип переменной определяется автоматически на основе присваиваемого этой переменной значения. Номинативность (nominal type system) языка касается системы типов и означает, что две переменные считаются эквивалентными, только если их типы данных в точности совпадают (наименование типа данных одинаково). В языке Julia номинативность проявляется при определении функций с явным указанием типов принимаемых ею аргументов. Компилятор выдаст ошибку, если передать такой функции в качестве аргумента переменную с отличным от требуемого типом данных.

Параметрический полиморфизм означает возможность объявлять функции с одинаковыми именами, но с разным числом аргументов или с аргументами иного типа данных. Такой подход способен заменить объектно/ориентированных подход к программированию, хотя и отличается от него идеологически.

Первая публичная версия языка была представлена в 2012 году, на начало 2017 года актуальной является версия 0.5. Язык реализован непосредственно на `Julia`, а также на `C`, `C++` и `Scheme`. Стандартная библиотека также почти полностью написана на `Julia` с несколькими зависимостями на языке C и Fortran (BLAS).

Перед тем как перейти к описанию конкретных аспектов языка, кратко перечислим основные идеи, вложенные в язык Julia.

Julia Lang нацелен именно *на научное программирование*. В связи с этим синтаксис языка максимально приближен к обозначениям, являющимся стандартными в математике. Например, нумерация элементов массива начинается с 1, а не с 0, что более привычно при записи индексов векторов и матриц. Также в язык встроена нативная реализация комплексных чисел, матричных операций, рациональных дробей, большое число встроенных математических функций. В язык встроена поддержка кодировки Unicode, поэтому становится возможным использовать в именах переменных практически любые символы из любого алфавита, поддерживаемого этой кодировкой.

Язык Julia имеет *простой и емкий синтаксис*, взявший много идей у популярных интерпретируемых языков. Базовый синтаксис `Julia` интуитивен и сравним по краткости с `Python`. Присутствует большое количество «синтаксического сахара», то есть дополнительных конструкций, упрощающих запись часто встречающихся выражений. Так в логических выражениях можно записывать цепочку сравнений (например, $1 < x < 3$) вместо того, чтобы соединять отдельные логические условия операторами логического «И» или «ИЛИ» (`(1 < x) && (x < 3)`).

Язык Julia изначально разрабатывался с прицелом на достаточно *высокую производительность*. Этого удалось достичь благодаря использованию LLVM (**L**ow **L**evel **V**irtual **M**achine) компиляции. В некоторых задачах `Julia` приближается по скорости выполнения к компилируемым языкам, таким как C и Fortran, хотя в целом не превосходит их. Другие же языки, чаще всего используемые в научных вычислениях, такие как `Matlab`, `Python`, `Mathematica`, `Maple`, уступают `Julia` на порядок. Также авторы языка предусмотрели прямой вызов библиотек на `C` и `Fortran`, что с одной стороны призвано компенсировать недостаток в производительности на критических участках программы, а с другой — восполнить отсутствие необходимых библиотек.

Многие системы для математических вычислений позволяют исполнять блоки кода *в интерактивном режиме*. Такая возможность актуальна при использовании языка в научных исследованиях и при обработке данных, так как очень часто

алгоритм работы программы может изменяться в процессе исследования и в зависимости от решаемой задачи. Julia может работать как в режиме интерактивной командной строки, так и использовать Jupyter для отображения графиков и форматированных данных.

В связи с широкой доступностью многопроцессорных компьютеров и небольших кластеров в язык Julia изначально вложена поддержка параллельных и распределенных вычислений.

3. Множественная диспетчеризация

Отдельно стоит упомянуть о *множественной диспетчеризации* (multiple dispatch), которая реализует упомянутый выше параметрический полиморфизм и позволяет создать множество реализаций одной функции, охватывающих различные комбинации типов и количества аргументов. После этого компилятор автоматически может определить, какую из реализаций функции использовать в том или ином случае.

Рассмотрим небольшой пример, иллюстрирующий эту особенность языка Julia.

```
function f1(x::Real)
  println("Real")
end

function f1(x::String)
  println("String")
end
```

За функцией `f1` будет закреплено два метода. Первый метод принимает в качестве аргумента действительное число и только его, а второй метод — только строку. В ходе выполнения кода выбор необходимого метода происходит автоматически.

```
> f1(10.0)
Real

> f1("Строка")
String
```

Большинство встроенных функций используют множественную диспетчеризацию для того, чтобы корректно работать с аргументами различных типов. Например, функция *sqrt*() имеет 10 различных методов для обработки комплексных и действительных чисел различной разрядности, целых чисел, а также чисел с бесконечной точностью типа `BigFloat` и `BigInt`.

Кроме функций с одинаковыми именами, но с аргументами разных типов, можно определять одноименные функции с разным числом аргументов. Например:

```
function f1(x::Real, y::Int)
  println("x = Real, y = Int")
end

function f1(x::Real, y::Complex)
  println("x = Real, y = Complex")
end
```

При компиляции данного кода, к двум уже существующим методам функции `f1` будут добавлены еще два:

```
> f1(10.0, 2)
x = Real, y = Int

> f1(2.0, 1.0im)
x = Real, y = Complex
```

Однако попытка вызова функции `f1` с двумя комплексными аргументами приведет к ошибке:

```
> f1(1im, 1im)
MethodError: no method matching f1(::Complex64, ::Complex64)
```

Julia поддерживает задание функций со значениями аргументов по умолчанию, а также с необязательными аргументами, однако эти возможности представляют по сути синтаксический сахар, так как внутренне реализуются при помощи множественной диспетчеризации.

4. Параллельные вычисления

В настоящий момент существуют несколько широко доступных аппаратных решений, которые обеспечивают выполнение параллельных программ: многопроцессорные системы с общей памятью, кластеры и системы с графическими ускорителями (CUDA и OpenCL). В язык Julia встроены средства для написания программ, предназначенных для выполнения на кластере, а также на одном компьютере с многоядерным процессором. Параллелизм в Julia в основном реализован в рамках процессов, а нити (потоки) поддерживаются пока только экспериментально.

В отличие от технологии MPI, параллельное программирование в Julia опирается не на явное получение и отправление сообщений от процесса к процессу, а на *удаленные обращения* (remote references) и *удаленные вызовы* (remote calls). Удаленное обращение может использоваться любым процессом для получения доступа к любому объекту, находящемуся в ведении другого процесса. Удаленный вызов в свою очередь позволяет одному процессу послать запрос другому процессу на выполнение определённой функции. Распределение работы между процессами в большинстве случаев осуществляется автоматически. Удаленный вызов производится с помощью функции `remotecall()`. Данная функция немедленно возвращает объект Future, хотя вычисления могут еще долго продолжаться на удаленном процессе. Для получения результатов вычисления с удаленного процесса применяется функция `fetch()` или `wait()`. Удаленный вызов также может осуществлять макрос `@spawnat`. Функция `remotecall_fetch()` является по своей сути комбинацией функции `remotecall()` и `fetch()`, то есть процесс выполнения на текущем процессе останавливается до получения результата вычисления с удаленного процесса.

Для работы с кластерами в Julia предусмотрены соединение с удаленными машинами с помощью ssh, а также управление запущенными на удалённых узлах процессами. Обмен данными между процессами в большинстве случаев происходит неявно, что отличается от подходя технологии MPI, где программист должен прописать логику передачи сообщений между процессами при разработке кода.

Механизм разделяемой памяти реализуется с помощью общих массивов, которые инициализируются следующим конструктором:

```
SharedArray(T::Type, dims::NTuple; init=false, pids=Int[])
```

Массивы данного типа позволяют нескольким процессам работать с общими данными. Далее мы рассмотрим небольшой пример их использования.

5. Параллельные циклы и map reduce

Многие задачи, которые можно эффективно распараллелить, не требуют интенсивной пересылки данных. К таким задачам относится параллельное выполнение различных витков цикла. В Julia предусмотрены макросы для этой цели.

Рассмотрим следующие простой пример:

```
nheads = @parallel (+) for i=1:200000000
  Int(rand(Bool))
End
```

Итерации данного цикла распределяются между процессами, затем к результатам работы процессов применяется операция, указанная в скобках после макроса `@parallel` (в нашем примере +). Таким способом можно очень эффективно распараллеливать циклы, итерации которых могут выполнятся независимо друг от друга.

Если в параллельном цикле необходимо манипулировать с массивом данных, то вместо стандартного массива следует использовать разделяемый массив (SharedArray). Элементы такого массива будут доступны всем задействованным процессам, и его можно применять в параллельных циклах:

```
a = SharedArray(Float64,10)
@parallel for i=1:10
  a[i] = i
end
```

6. Параллелизация с помощью нитей

Julia поддерживает параллелизацию с помощью нитей, хотя на данный момент эта функция считаются экспериментальной и синтаксис может существенно измениться по мере развития языка. В отличие от процессов нити имеют общую область памяти и требуют меньше ресурсов. Нити оптимальны при параллелизации в рамках одного многоядерного процессора. На данный момент имеющийся функционал обеспечивает макрос `@threads`. Он, например, позволяет распределить итерации цикла между нитями:

```
Threads.@threads for i = 1:10
  a[i] = Threads.threadid()
end
```

К сожалению, поддержка нитей в Julia пока весьма ограничена по сравнению, например, с OpenMP.

7. Заключение

На данный момент язык Julia поддерживает базовые технологии параллельных вычислений на основе процессов. Синтаксис, обеспечивающий написание параллельных программ, встроен в ядро языка, поэтому есть уверенность, что он будет развиваться по мере развития языка. На данный момент имеющегося функционала уже достаточно для создания производительных параллельных программ, работающих как на кластере, так и на одном компьютере с несколькими процессорами (ядрами).

Литература

1. Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah, Julia: A Fresh Approach to Numerical Computing (2014), November. arXiv : cs.MS/1411.1607.
2. Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman, Julia: A Fast Dynamic Language for Technical Computing (2012), September. arXiv : cs.PL/1209.5145.
3. Malcolm Sherrington Mastering Julia. Packt Publishing, 2015.