УДК 004.272
DOI:10.17853/2587-6910-2021-04-81-86

НАУЧНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JULIA: РАСПАРАЛЛЕЛИВАНИЕ ФРАГМЕНТА КОДА ДЛЯ УЛУЧШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

SCIENTIFIC PROGRAMMING IN JULIA: PARALLELIZING A PIECE OF CODE TO IMPROVE PERFORMANCE

Евгения Александровна Оконешникова

студентка факультета математики и компьютерных наук, 5-й курс

owljaneok@yandex.ru

ФГБОУ ВО «Кубанский государственный университет», Краснодар, Россия

Evgeniya Aleksandrovna Okoneshnikova

Kuban State University, Krasnodar, Russia

Аннотация. *Приводится решение задачи математической физики средствами языка программирования Julia как пример применения данного языка в учебном процессе и научных исследованиях. Основное внимание обращено на ускорение вычислений путем распараллеливания части кода.*

Ключевые слова: *научное программирование, язык программирования Julia, параллельное программирование, построение численного решения задач математической физики.*

Abstract. *The article provides a solution to the problem of mathematical physics using the Julia programming language, as an example of the use of this language in the educational process and scientific research. The focus is on speeding up computations by parallelizing a piece of code.*

Keywords: *scientific programming, Julia programming language, parallel programming, construction of numerical solutions to problems of mathematical physics.*

Задачи научного программирования, как правило, отличаются трудоемкими, масштабными вычислениями, могут быть связаны с анализом большого объема данных. Исследователи, зачастую не являющиеся высококвалифицированными программистами, натыкаются на препятствие в виде необходимости погружения в синтаксис и семантику выбранного языка программирования для достижения оптимальной производительности.

Объектом изучения в данной статье выступает высокопроизводительный язык Julia — текущая версия 1.5.3 [julialang.org]. Julia обладает простым синтаксисом в духе Python и MatLab и богатым набором математических пакетов. Вот лишь некоторые из них: пакет компьютерной алгебры Nemo и линейной алгебры LinearAlgebra, экосистема дифференциальных уравнений DifferentialEquations, семейство пакетов для оптимизации и решения уравнений JuliaNLSolvers, эффективные алгоритмы итерационного решения линейных систем IterativeSolvers, пакет статистического анализа StatsBase, быстрые преобразования Фурье AbstractFFTs. К сильным сторонам этого языка можно отнести поддержку параллельных вы-

числений и работу с большими массивами данных. Все перечисленное делает язык Julia привлекательным как для проведения научных исследований, так и для применения в учебном процессе.

На примере решения задачи математической физики сравним эффективность последовательного и параллельного кода программы.

Рассмотрим численное решение задачи о колебаниях линейно упругого изотропного слоя толщиной H с нагрузкой $q(x_1)$ и нестационарным импульсом $p(t)$. Зависящие от времени смещения слоя обычно выражаются через гармонические колебания (1), к которым применяется обратное преобразование Фурье [1].

$$u(x_1, x_2, \omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \frac{e^{-\sqrt{\alpha^2-k(\omega)^2}x_2}\left(-1+e^{2\sqrt{\alpha^2-k(\omega)^2}(H+x_2)}\right)Q(\alpha)}{\left(1+e^{2H\sqrt{\alpha^2-k(\omega)^2}}\right)\sqrt{\alpha^2-k^2}} e^{-i\alpha x_1}\, d\alpha$$

$$= i\sum_{m=1}^{\infty} res\ K(-\zeta_m, x_2, \omega)Q(-\zeta_m)e^{i\zeta_m x_1}, \quad x_1 > 0 \qquad (1)$$

Тогда решение задачи $u(x_1,x_2,t)$ можно записать в виде интеграла (2).

$$u(x_1, x_2, t) = \frac{1}{\pi} Re \int_0^{\infty} u(x_1, x_2, \omega)P(\omega)e^{-i\omega t}\, d\omega \qquad (2)$$

В качестве входных данных задаем координаты точки, толщину слоя, плотность, упругую постоянную Ламе, а также функции $p(t)$ и $q(x1)$. Определяем диапазон частот $\omega \in [-\omega_1;\omega_1]$ как носитель преобразования Лапласа от $p(t)$, т. е. должно выполняться следующее условие: $\forall\ \omega > \omega_1 P(\omega) << 1$. Разделим этот диапазон на 20 тысяч равноотстоящих значений.

Вычислим интеграл (1) для $\omega \in [-\omega_1;\omega_1]$, применяя теорию вычетов. Заметим, что при суммировании можно не учитывать вычеты по чисто мнимым полюсам. Из подынтегрального выражения выделим часть, зависящую не от частоты ω а только от полюса ζ_m — весовая функция и экспонента. Оставшуюся часть представим в виде функции res. А затем просуммируем элементы ряда с помощью процедуры reduce. Данный фрагмент кода приведен на рис. 1.

Здесь знак точки задает вектор-функцию от переданного вектора-аргумента. А оператор .* означает покомпонентное произведение соответствующих массивов.

Теперь можем вычислить значения функции $u(x_1,x_2,\omega)$ по формуле (1) для выбранного диапазона частот. Эта часть оказывается наиболее времязатратной. С помощью макроса @time узнаем, сколько требуется времени и памяти для вычисления массива u_om (рис. 2).

Повторив эту процедуру еще четыре раза, вычислим среднее время, необходимое для выполнения расчетов:

$$t_{ср.послед.} = \frac{180.9331 + 178.823 + 177.8254 + 172.521 + 188.4791}{5} = 179.716\ с.$$

Вычисление интеграла (2) занимает считанные секунды, после чего можем построить график решения $u(x_1,x_2,t)$ (рис. 3).

Итак, на данном этапе у нас есть корректно работающий последовательный код. Но процесс вычисления $u(x_1,x_2,\omega)$ для массива из 20 тысяч значений частот занимает около трех минут. Его можно ускорить, прибегнув к методам параллельной обработки.

Поддержку параллельных вычислений в Julia обеспечивает пакет Distributed, который является частью стандартной библиотеки. Julia — язык с «односторонней» коммуникацией, т. е. для выполнения двухпроцессорной операции программисту достаточно управлять явно только одним процессом.

У каждого процесса есть связанный идентификатор. Процессы, используемые по умолчанию для параллельных операций, называются «рабочими». Добавить n «рабочих» процессов можно

```
res(a,k,x2) = exp(-a(a, k)*x2)*(-1+exp(2*a(a, k)*(H+x2)))/((2*H*a+a/
a(a, k))*exp(2*H*a(a, k))+a/a(a, k))

QE_p = Q.(-1*pol) .* exp.(x1*pol*im)

u(x1,x2,ω) = im * reduce(+, res.(-1*pol,k(ω),x2).*QE_p)
```

Рис. 1. Задание вспомогательных функций
и функции гармонических колебаний

```
julia> @time u_om = u.(x1,x2,om)
180.933146 seconds (3.21 G allocations: 98.370 GiB, 2.49% gc time)
20000-element Array{Complex{Float64},1}:
 8.219185137082958 + 2.5120294303055895im
  8.35551473626018 + 2.5978762323999116im
 8.490963962724747 + 2.6850514202404887im
 8.625520185154599 + 2.773545834567769im
 8.759170869938838 + 2.8633502022694675im
 8.891903582078445 + 2.954455137466857im
 9.023705986069452 + 3.046851142603l6im
 9.154565846775593 + 3.1405286095374265im
 9.284471030289943 + 3.2354778206429744im
 9.413409504780812 + 3.3316889499082385im
                   ⋮
```

Рис. 2. Вычисление вектора-значений функции колебаний
с применением макроса @time для фиксации времени исполнения

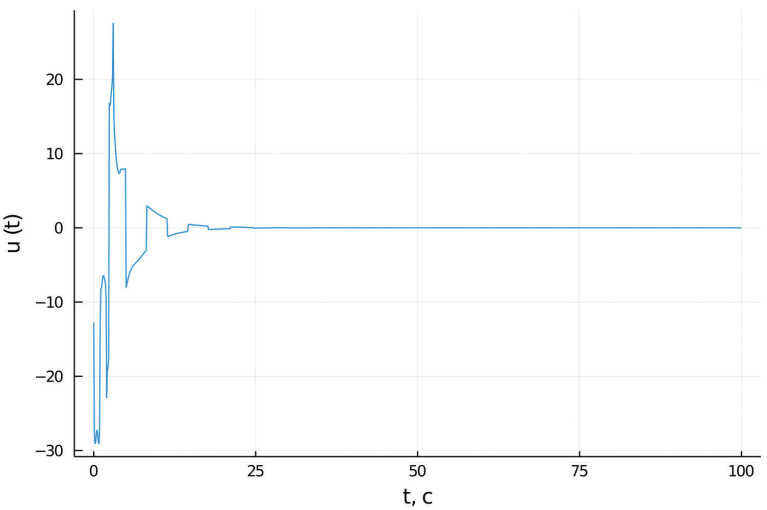

Рис. 3. График функции, описывающей смещения слоя

```
function u_prl(x1,x2,ω,m,pol,QE_p)
    Sum = @distributed (+) for i=1:length(pol)
        res(-1*pol[i],k(ω,m),x2,H)*QE_p[i]
    end
    return im*Sum
end

@time u_om1 = map(ω->u_prl(x1,x2,ω,μ/ρ,pol,QE_p), om)
```

Рис. 4. Пример распараллеливания функции
гармонических колебаний

```
julia> using LinearAlgebra

julia> norm(u_om.-u_om1)
5.722612904204765e-12
```

Рис. 5. Вычисление нормы разности векторов

с помощью функции addprocs(n), причем, чтобы процессам не пришлось конкурировать за управление потоком исполнения, их количество не должно превышать количество ядер процессора вычислительной машины. В нашем случае работа выполняется на бытовом ПК, с четырехъядерным процессором Intel Core i3-6006U CPU 2.00GHz × 4. Поэтому имеет смысл добавить к главному процессу не более трех «рабочих».

При написании параллельного кода нужно следить за тем, чтобы функции, вызываемые главным процессом, были определены на «рабочих». Для задания функции сразу на всех процессах можно воспользоваться макросом @everywhere. Если вызванная на «рабочем» процессе функция предполагает выполнение операций с некоторыми данными, то либо все необходимые переменные должны быть определены на самом «рабочем» процессе, либо ему должны быть переданы ссылки на них. Наряду с этим, обмен данными можно осуществлять с помощью каналов Channel [3, 4, 5]. Для экономии времени следует избегать пересылки больших объемов данных между процессами. Если предполагается обработка данных одного массива несколькими процессами, то его следует определить как общий массив — SharedArray — тип, поставляемый с пакетом SharedArrays.

Перед нами стоит задача найти способ распараллеливания функции $u(x_1,x_2,\omega)$, которая сопоставляет частоте om[i]=ω_i сумму вычетов подынтегрального выражения. Множество полюсов распределим между рабочими процессами, чтобы они независимо друг от друга нашли вычеты, а главный процесс будет производить их суммирование. Реализовать это можно с помощью макроса @distributed (op), который указывается перед циклом и распределяет его ите-

рации между всеми доступными процессами. Указанная в скобках операция op собирает результат параллельных вычислений. Полученную функцию назовем u_prl(x1, x2, ω, m, pol, Q_p, E_p). Все данные, необходимые для вычислений, нужно передать в качестве аргументов. Вспомогательные функции должны быть определены на каждом рабочем процессе. Эту функцию нужно применить ко всем элементам массива частот om. Процедура map(f, col) применяет функцию f к массиву col поэлементно. Данный фрагмент кода приведен на рис. 4.

В результате получим массив данных — вектор значений функции $u(x_1,x_2,\omega)$, который для параллельного случая назовем u_om1.

Чтобы убедиться в равенстве полученных векторов u_om и u_om1, найдем норму их разности. Для этого подключим пакет линейной алгебры и воспользуемся функцией norm(A, p::Real=2),

```
julia> @time u_om1 = map(ω->u_prl(x1,x2,ω,μ/ρ,pol,QE_p), om)
 80.937026 seconds (8.67 M allocations: 362.226 MiB, 0.09% gc time)
20000-element Array{Complex{Float64},1}:
 8.219185137082956 + 2.51202943030559im
 8.355514736260169 + 2.5978762323999045im
 8.490963962724763 + 2.685051420240482im
 8.625520185154596 + 2.773545834567764im
 8.759170869938846 + 2.863350202269467im
 8.891903582078411 + 2.9544551374668564im
 9.023705986069489 + 3.046851142603157im
 9.154565846775597 + 3.140528609537423im
 9.284471030289968 + 3.235477820642976im
                   ⋮
```

Рис. 6. Параллельные вычисления значений функции колебаний с применением макроса @time

вычисляющей p-норму от вектора (или матрицы) A. По умолчанию параметр p равен 2 — евклидова норма. В нашем примере полученное значение близко к нулю (рис. 5), поэтому можем говорить об идентичности результатов при последовательном и при параллельном вычислениях.

С помощью макроса @time узнаем, сколько потребовалось времени и памяти для вычисления массива u_om1 (рис. 6).

Совершив пять повторений, установим среднее время, необходимое для выполнения функции u_prl.

Ниже приведен полный листинг программы, где блок с вычислением значений $u(x_1,x_2,\omega)$ указан в двух вариантах: последовательном и параллельном.

$$t_{ср.парал.} = \frac{80.9370 + 78.3115 + 77.8012 + 83.3550 + 84.8905}{5} = 81.05904 \text{ с.}$$

Ниже приведен полный листинг программы, где блок с вычислением значений

Сравним время, необходимое для построения векторов-значений u_om и u_m1 для 20000 значений частот:

$$t_{ср.послед.} = 179.7\text{с. и } t_{ср.парал.} = 81\text{с.}$$

```
#Входные данные: точка, толщина слоя, плотность, константа Ламе
x1, x2 = 5, 0;
H, ρ, μ = 1, 1,1;
P(ω) = -(exp(ω*im)-1)*im/ω #Преобразование Лапласа от p(t)
Q(α) = 2/α*cos(α)*(1/α-im) #Преобразование Фурье от q(x1)
#Задание массива частот
ω1 = 50
om = collect(LinRange(-ω1, ω1, 20000));
k(ω) = ω*(μ/ρ)^0.5 #волновое число
a(α, k) = (α^2 — k^2)^0.5
#Задание массива полюсов
n=10;
poles(k) = Complex(k^2-(pi*(n-0.5)/H)^2)^0.5
pol = poles.(k.(om))
#Оставляем полюса только с ненулевой вещ.частью
filter!(i→real(i)>0.0,pol)
#часть, не зависящая от частоты
QE_p = Q.(-1*pol) .* exp.(x1*pol*im)
```

```
# Последовательные вычисления
res(α,k,x2) = exp(-a(α, k)*x2)*(-
1+exp(2*a(α, k)*(H+x2)))/((2*H*α+α/a(α,
k))*exp(2*H*a(α, k))+α/a(α, k))

u(x1,x2,ω) = im * reduce(+, res.
(-1*pol,k(ω),x2).*QE_p)
@time u_om = u.(x1,x2,om)
uP_om = u_om.*P.(om)
```

```
# Параллельное вычисления
using Distributed
addprocs(3)
#Определяем вспомогательные функции сразу на всех процессах
@everywhere k(ω,m) = ω*m^0.5
@everywhere a(α, k) = (α^2 - k^2)^0.5
@everywhere res(α,k,x2,H) = exp(-a(α, k)*x2)*(-1+exp(2*a(α,
k)*(H+x2)))/((2*H*α+α/a(α, k))*exp(2*H*a(α, k))+α/a(α, k))

function u_prl(x1,x2,ω,m,pol,QE_p)
 Sum = @distributed (+) for i=1:length(pol)
  res(-1*pol[i],k(ω,m),x2,H)*QE_p[i]
 end
 return im*Sum
end
@time u_om1 = map(ω→u_prl(x1,x2,ω,μ/ρ, pol, QE_p), om)
uP_om = u_om1.*P.(om)
```

```
#норма разности
using LinearAlgebra
norm(u_om.-u_om1)
#Интеграл от непрерывной функции по данному массиву точек
function Integral(t::Array, f::Function)
        f_k = f(t);
        d=t[2]-t[1];
        I = 0.5*(f_k[1]+f_k[end]) + reduce(+,f_k[2:end-1]);
 return I*d
end
fut(x,t) = uP_om.*exp.(-im*x*t) #подынтегральная функция (2)
u_t(x1,x2,t)=Integral(om, x->fut(x,t))/pi
t_n = collect(LinRange(0, 100, 1000));
u_tn = u_t.(x1,x2,t_n)
sgmod(x:: Complex) = sign(real(x))*abs(x);
using Plots
gr()
plot(t_n, sgmod.(u_tn),legend=false, xlabel = "t, c", ylabel = "u (t)")
```

Видно, что распараллеливание функции частотного спектра позволило более чем в два раза ускорить вычисления. Конечно, это не единственный, но синтаксически достаточно простой способ распределения задач между процессами. Двукратное уменьшение времени выполнения этого фрагмента кода позволило сократить общее время вычислений примерно на 35 %.

Список литературы

1. *Glushkov, E.* Lamb wave excitation and propagation in elastic plates with surface obstacles: proper choice of central frequencies / E. Glushkov, N. Glushkova, R. Lammering, A. Eremin, M. N. Neumann. Text: print // Smart Materials and Structures. 2010. Vol. 20, № 1.
2. *Marano Maza, M.* Lecture Notes: Distributed and parallel systems, Department of Compute Science / M. Marano Maza, Western University. 2017. Text: print.
3. *Документация* по языку Julia: официальный сайт. MIT, 2020. URL: https://docs.julialang.org/en/v1/. Текст: электронный.
4. *Оконешникова, Е. А.* Новый язык программирования в учебном процессе и научных исследованиях / Е. А. Оконешникова. Текст: электронный // Новые информационные технологии в об-

разовании и науке НИТО 2020: материалы 13-й Международной научно-практической конференции. 2020. С. 123–129.

5. *Оконешникова, Е. А.* Использование параллелизма в Julia на примере функции Эйлера / Е. А. Оконешникова. Текст: электронный // Информационные технологии в математике и математическом образовании: материалы 9-й Всероссийской с Международным участием научно-методической конференции. Красноярск, 12–13 ноября 2020 г. / отв. ред. В. Р. Майер; Краснояр. гос. пед. ун-т им. В. П. Астафьева. Красноярск, 2020. С. 34–38. URL: верстка_Майер_2020.indd (kspu.ru).