МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«УДМУРТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Институт математики, информационных технологий и физики
Кафедра вычислительной механики

КУРСОВАЯ РАБОТА

Реализация решения СЛАУ методом сопряжённых градиентов в Julia

Выполнил:
студент IV курса группы ОАБ-01.03.02-42 Филиппов Д. В.

Научный руководитель:
доктор наук, профессор Копысов С. П.

Ижевск – 2020 г.

Содержание

Введение

Одной из основных задач вычислительной линейной алгебры является задача решения систем линейных алгебраических уравнений (далее - СЛАУ). Решение СЛАУ является элементарной частью алгоритма при решении нелинейных задач.

Вычисления, связанные с решением СЛАУ, применяются во многих сферах человеческой деятельности. Во многих практических задачах затраты и прибыль линейно зависят от количества приобретенных или утилизированных средств. Например, суммарная стоимость партии товаров линейно зависит от количества закупленных единиц, а оплата за перевозки осуществляется пропорционально весу груза, который перевозится. На практике СЛАУ используются для решения проблем связанных с распределением ресурсов, планированием производства, организации работы транспорта, а так же при решении прикладных задач по физике, радиофизике, электронике и других областей науки и техники.

Возможность математического моделирования разных процессов с применением ЭВМ зависит от умения эффективно решать СЛАУ. В связи с этим уделяется особое внимание разработке и исследованию методов решения СЛАУ. Для решения СЛАУ используются прямые и итерационные методы.

Один из итерационных методов – метод сопряжённых градиентов (метод Флэтчера-Ривза). Он позволяет найти локальный экстремум функции на основе информации о её значениях и градиенте. Для решения СЛАУ используется его некая модификация.

Метод сопряжённых градиентов для решения СЛАУ

Метод сопряжённых градиентов для решения СЛАУ – численный метод решения систем алгебраических уравнений, является итерационным методом. Суть заключается в следующем – пусть дана система: $Ax=b$. То есть, мы имеем некоторую матрицу *A* и вектор *b*. Тогда процесс решения СЛАУ можно представить как минимизацию следующего функционала:

$$(Ax,x)-2(b,x)\to min \quad ,$$

где за $(:,:)$ обозначено скалярное произведение. Минимизируя данный функционал, используя подпространства Крылова, получаем алгоритм метода сопряжённых градиентов.

У метода сопряжённых градиентов для решения СЛАУ существуют некоторые ограничения – матрица системы должна быть вещественной, симметричной, положительно определённой: $A=A^T>0, A\in\mathbb{R}$.

Перед выполнением итерационной части метода нужно произвести подготовку:

1. выбрать начальное приближение x_0 ;
2. установить вектор $r_0=b-Ax_0$;
3. установить вектор $z_0=r_0$.

Рассмотрим алгоритм метода для *i*-ой итерации:

1. $\alpha_i=\frac{(r_{(i-1)},r_{(i-1)})}{(Az_{(i-1)},z_{(i-1)})}$
2. $x_i=x_{(i-1)}+\alpha_i z_{(i-1)}$
3. $r_i=r_{(i-1)}-\alpha_i Az_{(i-1)}$
4. $\beta_i=\frac{(r_i,r_i)}{(r_{(i-1)},r_{(i-1)})}$
5. $z_i=r_i+\beta_i z_{(i-1)}$

Поскольку минимизируемый функционал квадратичный, то алгоритм должен дать ответ на *n*-ой итерации (где *n* – размерность матрицы *A*). Но при реализации алгоритма на компьютере существует погрешность представления вещественных чисел, в результате чего может понадобиться больше итераций. Оценивать точность можно по относительной невязке и прекращать процесс, если невязка будет меньше заданного числа:

$$\frac{(\|r_i\|)}{(\|b\|)}<\epsilon \quad .$$

Постановка задачи

Задача заключается в реализации алгоритма метода сопряжённых градиентов (для решения СЛАУ). Реализовать данный алгоритм требуется на языке программирования Julia. Данный язык программирования очень хорошо подходит для решения математических задач.

Непосредственно сам алгоритм реализовать несложно. Но на практике мы часто имеем дело с разреженными матрицами – в них содержится много нулевых элементов. Для хранения таких матриц и векторов придуманы несколько форматов. Один из самых популярных – MatrixMarket (.mtx). Но, к сожалению, Julia с данным форматом не работает без применения сторонних библиотек. Зато она хорошо работает с другим форматом – csc. Файлы такого формата имеют вид:

$$\begin{matrix} 1 & 1 & 2 & 2 & 3 \\ 1 & 3 & 2 & 3 & 1 \\ 5 & 6 & 7 & 8 & 9 \end{matrix} .$$

В первой строке хранятся номера столбцов, во второй – номера строк, в третьей – значения. Таким образом, матрица из данного примера выглядит так:

$$\begin{pmatrix} 5 & 0 & 6 \\ 0 & 7 & 8 \\ 9 & 0 & 0 \end{pmatrix} .$$

Аналогичным образом хранится и вектор – для него, соответственно, нужны лишь две строки – в первой – позиции элементов, во второй – значения.

Так же для корректной работы алгоритма нужно убедиться, что матрица действительно соответствует входным ограничениям.

Пример реализации поставленной задачи на Julia

Для того, чтобы решить поставленную задачу, нам необходимо реализовать несколько вспомогательных функций.

Начнём сначала, а именно – с чтения матриц и векторов из файла, так как во-первых, это удобно, во-вторых – универсально. Для этого были написаны функции:

```
# Чтение разреженного вектора из файла
function read_csc_vector(filename)
  file = open(filename, "r")
  colptr = parse.(Int, split(readline(file)))
  nzval = parse.(Float64, split(readline(file)))
  close(file)
  return sparsevec(colptr, nzval)
end

# Чтение разреженной матрицы из файла
function read_csc_matrix(filename)
  file = open(filename, "r")
  colptr = parse.(Int, split(readline(file)))
  rowval = parse.(Int, split(readline(file)))
  nzval = parse.(Float64, split(readline(file)))
  close(file)
  return sparse(colptr, rowval, nzval)
end
```

На вход подаётся название файла, который нужно прочитать. Затем файл открывается в режиме “только для чтения” во избежание его повреждения. Далее построчно считываем содержимое файла в соответствующие массивы. После того, как все данные прочитаны, закрываем файл и возвращаем результат – новый объект (матрица или вектор) с соответствующими параметрами конструктора. Нужно отметить, что для поддержки работы с разреженными матрицами и векторами, нужно подключить пакет SparseArrays (он уже предустановлен в Julia).

Небольшое отступление - так как найти матрицу в формате .mtx гораздо проще, то я написал конвертер на языке программирования Java для преобразования .mtx файлов .csc. Код конвертера будет представлен в приложениях.

Также нам понадобятся функции матрично-векторного и скалярного произведения. Хотя в Julia уже есть данные функции, мы будем использовать свои, так как их можно распараллелить, а так же сравним наши функции и стандартные функции Julia. Реализация:

```
# Скалярное произведение векторов
function s_multiply(_vector1, _vector2)
  result = Threads.Atomic{Float64}(0.0)
  mlen = min(length(_vector1), length(_vector2))
  Threads.@threads for i = 1:mlen
    if _vector1[i] == 0 || _vector2[i] == 0
      continue
    end
    Threads.atomic_add!(result, _vector1[i] * _vector2[i])
  end
  return result[]
end

# Умножение матрицы на вектор
function mv_multiply(_matrix, _vector)
  result = zeros(length(_vector))
  Threads.@threads for i = 1:length(result)
    result[i] = s_multiply(_matrix[i, :], _vector)
  end
  return result
end
```

Алгоритм вполне стандартный, за исключением макроса *Threads.@threads* и объекта *result* в функции скалярного произведения. Макрос используется перед циклом *for* и нужен для того, чтобы программа могла запустить цикл в несколько потоков. Объект *result* – атомарный, то есть потокобезопасный. В функции скалярного произведения также добавлена проверка – если один из элементов равен нулю, то текущая итерация прекращается. Это сделано для того, чтобы не терять время для доступа к атомарному объекту *result*. В функции матрично-векторного произведения атомарные объекты не используются, так как в этом нет необходимости – результат является вектором, и доступ к каждому из элементов будет происходить только в один поток.

В процессе реализации я столкнулся с проблемой – исходные тестовые матрицы были не полностью симметричны. Для этого я добавил проверку на симметричность матрицы:

```
# Проверка матрицы на симметричность
function is_matrix_symmetric(matrix)
  for i in 1:size(matrix, 1)
    for j in i+1:size(matrix, 1)
      if (matrix[i, j] != matrix[j, i])
        return false
      end
    end
  end
  return true
```

```
end
```

Логика работы функции довольно проста – сравниваем элементы из строки и столбца, двигаясь по диагонали – они должны быть равными. Если мы находим неравные элементы – значит, матрица несимметрична.

Теперь, когда все вспомогательные функции готовы, можно приступить непосредственно к реализации метода сопряжённых градиентов. Зададим точность константой: `const eps = 1e-6`, что соответствует числу 0.000001. Я написал две функции, реализующие метод сопряжённых градиентов. В первой функции используются стандартные реализации матрично-векторного и скалярного произведения, во второй – мои функции, описанные выше. Первая функция:

```
# Реализация метода сопряжённых градиентов
function gradients(matrix, vector)
  global vector_x = Array{Float64,1}(zeros(length(vector_b)))
  global vector_r = vector_b - (matrix_A * vector_x)
  global vector_z = vector_r
  global i = 1
  while (eps < (abs(norm(vector_r)) / abs(norm(vector_b))))
    local vector_xp = vector_x
    local vector_rp = vector_r
    local vector_zp = vector_z
    mv1 = matrix_A * vector_zp
    alpha = (transpose(vector_rp) * vector_rp) / (transpose(mv1) * vector_zp)
    global vector_x = vector_xp + alpha * vector_zp
    global vector_r = vector_rp - alpha * (mv1)
    beta = (transpose(vector_r) * vector_r) / (transpose(vector_rp) * vector_rp)
    global vector_z = vector_r + beta * vector_zp
    global i += 1
  end
  return vector_x
end
```

Поясню несколько моментов. Начальное приближение x_0 - нулевой вектор. Цикл повторяется, пока не будет достигнута необходимая точность (значение *eps* задано константой). Матрично-векторное произведение $Az_{(i-1)}$ вычисляется один раз для каждой итерации. Чтобы вычислить скалярное произведение векторов стандартными средствами Julia, необходимо транспонировать один из векторов функцией *transpose(vector)*. Префикс *global* обозначает, что нужно использовать уже имеющийся объект, а не создавать копию для каждой итерации. Префикс *local* обозначает, что нужно создать временный объект, в данном случае – для одной итерации (если префикс отсутствует и объект не определён ранее, то он считается локальным).

Вторая реализация алгоритма:

```
# Реализация метода сопряжённых градиентов с использованием параллельных вычислений
function gradients_parallel(matrix, vector)
  global vector_x = Array{Float64,1}(zeros(length(vector_b)))
  global vector_r = vector_b - mv_multiply(matrix_A, vector_x)
  global vector_z = vector_r
  global i = 1
  while (eps < (abs(norm(vector_r)) / abs(norm(vector_b))))
    local vector_xp = vector_x
    local vector_rp = vector_r
    local vector_zp = vector_z
    local mv1 = sparse(mv_multiply(matrix_A, vector_zp))
    alpha = s_multiply(vector_rp, vector_rp) / s_multiply(mv1, vector_zp)
    global vector_x = vector_xp + alpha * vector_zp
    global vector_r = vector_rp - alpha * mv1
    beta = s_multiply(vector_r, vector_r) / s_multiply(vector_rp, vector_rp)
    global vector_z = vector_r + beta * vector_zp
    global i += 1
  end
  return vector_x
end
```

Как можно заметить, данная реализация отличается от предыдущей только использованием других функций для вычисления матрично-векторного и скалярного произведения. Необходимо сказать, что для поддержки многопоточности необходимо запускать программу с ключом “*-t [число потоков]*”.

Для проверки работы нужно заранее определить все функции, задать точность *eps*, а так же добавить вывод некоторой информации (при необходимости). Приведу пример:

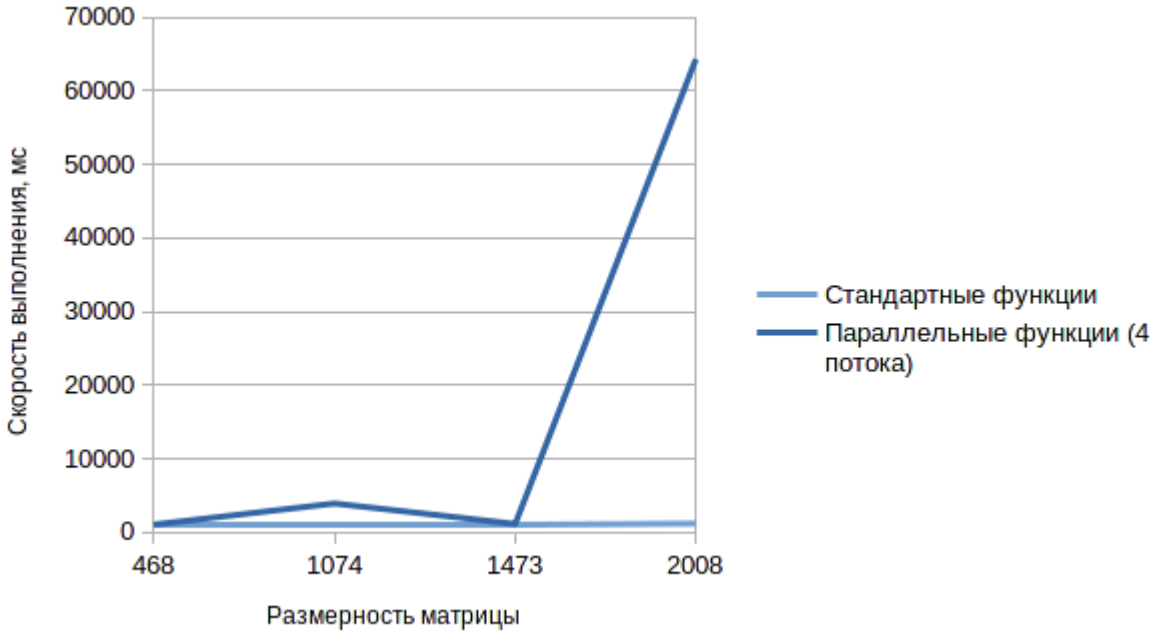
```
# Считываем матрицу и вектор
matrix_A = read_csc_matrix(matrix_file)
vector_b = read_csc_vector(vector_file)

# Проверяем матрицу на симметричность
if is_matrix_symmetric(matrix_A)
  # Запускаем алгоритм
  vector_x = gradients(matrix_A, vector_b)
  println(vector_x)
end
```

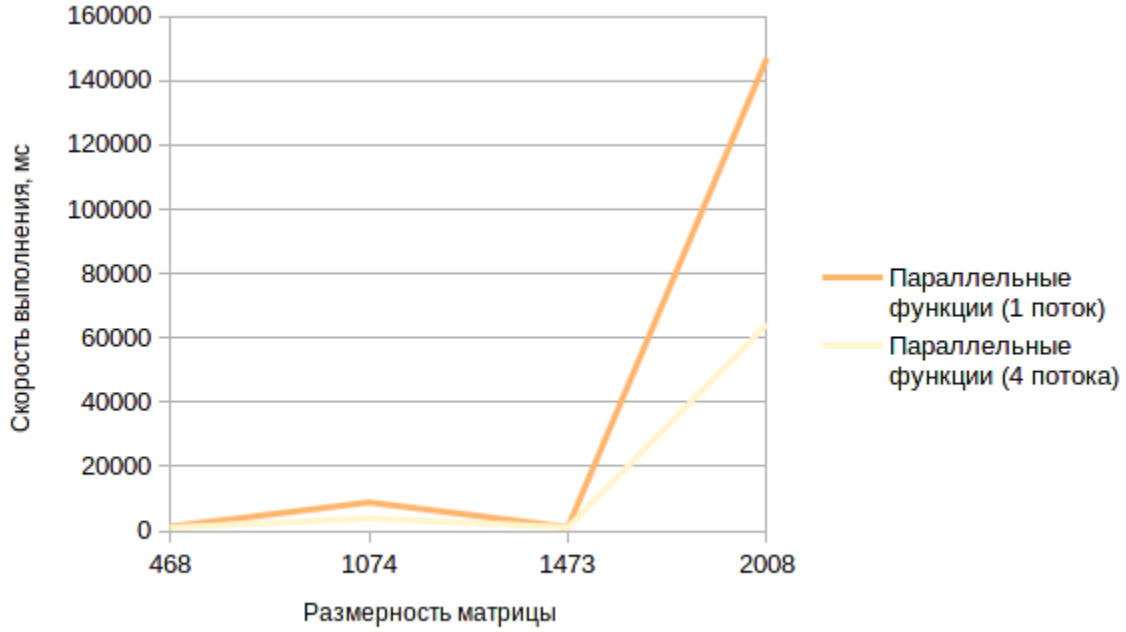
На этом реализация поставленной задачи завершена.

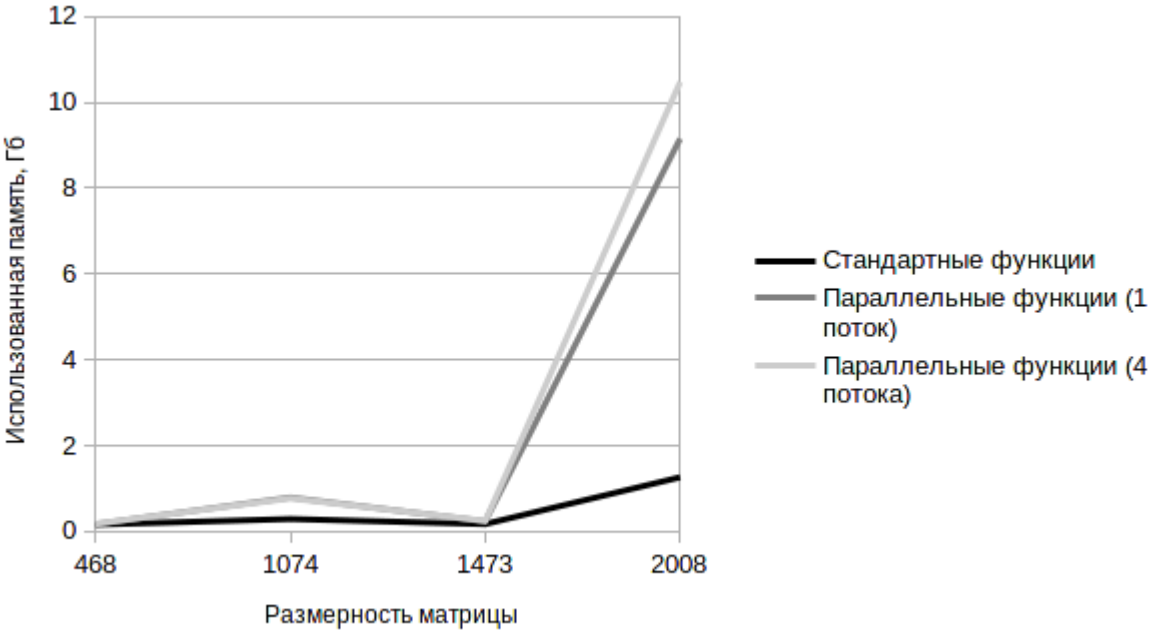
Тестирование, измерения и выводы

После написания кода и тестирования программ можно произвести некоторые измерения. Замеры были произведены на компьютере с операционной системой Manjaro Linux на ядре Linux версии 5.9.11, версия Julia – 1.5.2. Конфигурация компьютера – процессор Intel Core i3-8100 @3600 МГц, оперативная память DDR4, работающая в двухканальном режиме на частоте 2400 МГц с таймингами 17-17-17-39, объём 16 Гб, файл подкачки не используется. Программы запускались последовательно и в разных сессиях Julia во избежание кэширования некоторых данных и лишней нагрузки. Каждая функция была вызвана по 5 раз для каждого набора входных данных, и в таблицу вносилось среднее арифметическое значение от времени выполнения функции и количества использованной памяти (таблицы в текстовом виде можно найти в приложениях). Дополнительно в приложениях можно найти демонстрацию работы алгоритма и сходимости на небольших исходных матрице и векторе.

В тестировании и измерениях участвовало 4 набора данных – это матрицы с размерами 468*468 (2820 ненулевых элементов), 1074*1074 (7017 ненулевых элементов), 1473*1473 (17857 ненулевых элементов) и 2003*2003 (42943 ненулевых элемента), а так же векторы к ним, подобранные таким образом, что результатом решения является единичный вектор. Матрицы были взяты с сайта https://math.nist.gov/MatrixMarket и конвертированы в формат .csc. Векторы были получены путём матрично-векторного произведения исходной матрицы и единичного вектора, а затем результат был записан в файл .csc.







На данных графиках можно наглядно оценить скорость работы и использование оперативной памяти. Функции матрично-векторного и скалярного произведения, которые были написаны самостоятельно, в некоторых ситуациях работают медленнее (даже в несколько потоков) соответствующих стандартных функций Julia, так как стандартные функции имеют низкоуровневую оптимизацию. Такую же ситуацию можно наблюдать и с использованием оперативной памяти.

Также можно заметить, что скорость и использование памяти не зависит напрямую от начального вектора x_0 , размерности матрицы и её заполнения. Так происходит потому, что значения в исходных матрице и векторе оказались “удачными” для данного алгоритма, то есть для решения данной задачи потребовалось небольшое количество итераций.

Подведём итог измерений. При малых размерах исходных объектов, либо при “удачном” стечении обстоятельств, либо при выборе вектора x_0 , близкого к решению – разница между методами будет незначительной, так как количество итераций будет небольшим.

При больших размерах исходных объектов стандартные функции используют примерно в 8 раз меньше памяти, быстрее в 53 раза при выполнении в 4 потока и в 122 раза при выполнении в 1 поток. К сожалению, стандартные функции невозможно выполнять в несколько потоков – они работают только в одном потоке.

Можно сделать вывод, что для больших исходных объектов оптимально будет использование стандартных функций при использовании метода сопряжённых градиентов, следовательно, для компьютера будут важны следующие параметры – производительность центрального процессора на одно ядро, скорость кэша процессора и скорость чтения и записи в оперативную память.

Заключение

При реализации метода сопряжённых градиентов для решения СЛАУ мной были изучены некоторые возможности языка программирования Julia, рассмотрены методы распараллеливания программы, а так же я ознакомился с особенностями хранения и использования разреженных матриц и векторов. К сожалению, функции матрично-векторного и скалярного произведения не получилось ускорить относительно аналогичных стандартных функций. Тем не менее, рассмотренный метод сопряжённых градиентов можно считать достаточно эффективным для решения СЛАУ.

Список литературы

1. Ben Lauwens. Think Julia. / Ben Lauwens, Allen B. Downey. - O'Reilly Media, 2019. - 298 с.
2. Крис Касперски. Техника оптимизации программ. Эффективное использование памяти. / Крис Касперски. - "БХВ-Петербург", 2003. - 464 с.
3. Джон Клейнберг. Алгоритмы: разработка и применение. / Джон Клейнберг, Ева Тардос. - "Питер", 2016. - 800 с.
4. Захарченко Раиса. Использование методов решения систем линейных алгебраических уравнений в обучающем процессе ИТ-специалистов / Р. Н. Захарченко, Т. Г. Кирюшатова, Е. В. Кирюшатова / Проблеми інформаційних технологій. - 2016. - № 2. - С. 162-167.

Приложения

1. Ссылка на исходный код проекта – https://github.com/Damarreve/ConjugateGradients.

2. Исходный код программы на Julia.

```
using Dates
using SparseArrays
using LinearAlgebra

# ---------------------------------------------
#
# Реализация алгоритма решения СЛАУ методом сопряжённых градиентов.
#
# Матрица должна быть симметричной положительно определённой.
#
# ---------------------------------------------

# Требуемая точность
const eps = 1e-6

# Вывод процесса в stdout
const debug = false
# Вывод входных данных
const info = false
# Вывод итераций
const process = false

# Чтение разреженного вектора из файла
function read_csc_vector(filename)
  file = open(filename, "r")
  colptr = parse.(Int, split(readline(file)))
  nzval = parse.(Float64, split(readline(file)))
  close(file)
  return sparsevec(colptr, nzval)
end

# Чтение разреженной матрицы из файла
function read_csc_matrix(filename)
  file = open(filename, "r")
  colptr = parse.(Int, split(readline(file)))
  rowval = parse.(Int, split(readline(file)))
  nzval = parse.(Float64, split(readline(file)))
  close(file)
  return sparse(colptr, rowval, nzval)
end

# Скалярное произведение векторов
function s_multiply(_vector1, _vector2)
  result = Threads.Atomic{Float64}(0.0)
  mlen = min(length(_vector1), length(_vector2))
```

```
  Threads.@threads for i = 1:mlen
    if _vector1[i] == 0 || _vector2[i] == 0
      continue
    end
    Threads.atomic_add!(result, _vector1[i] * _vector2[i])
  end
  return result[]
end

# Умножение матрицы на вектор
function mv_multiply(_matrix, _vector)
  result = zeros(length(_vector))
  Threads.@threads for i = 1:length(result)
    result[i] = s_multiply(_matrix[i, :], _vector)
  end
  return result
end

# Вывод полученного вектора
function print_vector(vector)
  print("[")
  for i in vector_x[:]
    print(" ", (abs(i) < eps ? 0 : i))
  end
  println("]")
end

# Проверка матрицы на симметричность
function is_matrix_symmetric(matrix)
  for i in 1:size(matrix, 1)
    for j in i+1:size(matrix, 1)
      if (matrix[i, j] != matrix[j, i])
        return false
      end
    end
  end
  return true
end

# Превращение несимметричной матрицы в симметричную (копирование верхней правой
части в нижнюю левую)
function make_matrix_symmetric(matrix)
  for i in 1:(size(matrix, 1) - 1)
    for j in (i + 1):size(matrix, 1)
      if (matrix[i, j] != matrix[j, i])
        matrix[j, i] = matrix[i, j]
      end
    end
  end
end

# --------------
# Основная часть
# --------------
```

```
# 5x5
# matrix_file = "resources/little_matrix.csc"
# vector_file = "resources/little_vector.csc"

# 468x468
matrix_file = "resources/nos5.csc"
vector_file = "resources/nos5_vector.csc"

# 1074x1074
# matrix_file = "resources/bcsstk08.csc"
# vector_file = "resources/bcsstk08_vector.csc"

# 1473x1473
# matrix_file = "resources/bcsstk12.csc"
# vector_file = "resources/bcsstk12_vector.csc"

# 2003x2003
# matrix_file = "resources/bcsstk13.csc"
# vector_file = "resources/bcsstk13_vector.csc"

# Реализация метода сопряжённых градиентов
function gradients(matrix, vector)
  println("epsilon: ", eps)
  global vector_x = Array{Float64,1}(zeros(length(vector_b)))
  global vector_r = vector_b - (matrix_A * vector_x)
  global vector_z = vector_r

  if (info)
    println("vector_b: ", vector_b)
    println("matrix_A: ", matrix_A)
    println("vector_x0: ", vector_x)
    println("vector_r0: ", vector_r)
    println("vector_z0: ", vector_z)
  end

  global i = 1
  while (eps < (abs(norm(vector_r)) / abs(norm(vector_b))))
    local vector_xp = vector_x
    local vector_rp = vector_r
    local vector_zp = vector_z
    mv1 = matrix_A * vector_zp
    alpha = (transpose(vector_rp) * vector_rp) / (transpose(mv1) * vector_zp)
    if (debug) println("alpha[", i, "]: ", alpha) end
    global vector_x = vector_xp + alpha * vector_zp
    if (debug) println("vector_x[", i, "]: ", vector_x) end
    global vector_r = vector_rp - alpha * (mv1)
    if (debug) println("vector_r[", i, "]: ", vector_r) end
    if (debug) println("vector_rm[", i, "]: ", vector_rp) end
    beta = (transpose(vector_r) * vector_r) / (transpose(vector_rp) * vector_rp)
    if (debug) println("beta[", i, "]: ", beta) end
    global vector_z = vector_r + beta * vector_zp
    if (debug) println("vector_z[", i, "]: ", vector_z) end
    if (process)
      println("[#", i, "] alpha: ", alpha)
      println("[#", i, "] beta: ", beta)
```

```
      print("[#", i, "] vector x: ")
      print_vector(vector_x, eps)
      println()
    end
    global i += 1
  end

  println("Total iterations: ", i - 1)
  print("x: ")
  print_vector(vector_x)
end

# Реализация метода сопряжённых градиентов с использованием параллельных вычислений
function gradients_parallel(matrix, vector)
  println("epsilon: ", eps)
  global vector_x = Array{Float64,1}(zeros(length(vector_b)))
  global vector_r = vector_b - mv_multiply(matrix_A, vector_x)
  global vector_z = vector_r

  if (info)
    println("vector_b: ", vector_b)
    println("matrix_A: ", matrix_A)
    println("vector_x0: ", vector_x)
    println("vector_r0: ", vector_r)
    println("vector_z0: ", vector_z)
  end

  global max_ops = length(vector_b) * 2
  global i = 1
  while (eps < (abs(norm(vector_r)) / abs(norm(vector_b))))
    local vector_xp = vector_x
    local vector_rp = vector_r
    local vector_zp = vector_z
    local mv1 = sparse(mv_multiply(matrix_A, vector_zp))
    alpha = s_multiply(vector_rp, vector_rp) / s_multiply(mv1, vector_zp)
    if (debug) println("alpha[", i, "]: ", alpha) end
    global vector_x = vector_xp + alpha * vector_zp
    if (debug) println("vector_x[", i, "]: ", vector_x) end
    global vector_r = vector_rp - alpha * mv1
    if (debug) println("vector_r[", i, "]: ", vector_r) end
    if (debug) println("vector_rm[", i, "]: ", vector_rp) end
    beta = s_multiply(vector_r, vector_r) / s_multiply(vector_rp, vector_rp)
    if (debug) println("beta[", i, "]: ", beta) end
    global vector_z = vector_r + beta * vector_zp
    if (debug) println("vector_z[", i, "]: ", vector_z) end
    if (debug) println() end
    if (process)
      println("[#", i, "] alpha: ", alpha)
      println("[#", i, "] beta: ", beta)
      print("[#", i, "] vector x: ")
      print_vector(vector_x, eps)
      println()
    end
    global i += 1
  end
```

```
  println("Total iterations: ", i - 1)
  print("x: ")
  print_vector(vector_x)
end

matrix_A = read_csc_matrix(matrix_file)
vector_b = read_csc_vector(vector_file)

println("Матрица ", matrix_file)

if !is_matrix_symmetric(matrix_A)
  println("Корректируем матрицу - добиваемся симметрии")
  make_matrix_symmetric(matrix_A)
end

t_start = now()
@time gradients(matrix_A, vector_b)
t_end = now()
println("gradients(): ", t_end - t_start)

t_start = now()
@time gradients_parallel(matrix_A, vector_b)
t_end = now()
println("gradients_parallel(): ", t_end - t_start)
```

3. Таблицы с результатами измерений.

Размерность / скорость выполнения (мс)	**Стандартные функции**	**Параллельные функции (1 поток)**	**Параллельные функции (4 потока)**
468	929	1360	1047
1074	1008	8878	3923
1473	938	1192	1139
2008	1188	147022	64320

Размерность / использование памяти (Гб)	**Стандартные функции**	**Параллельные функции (1 поток)**	**Параллельные функции (4 потока)**
468	0.159	0.169	0.175
1074	0.29	0.791	0.772
1473	0.171	0.234	0.243
2008	1.263	9.153	10.47

4. Демонстрация сходимости.

```
epsilon: 1.0e-6
```

```
[#1] alpha: 0.090929203539823
[#1] beta: 0.06546093076983318
[#1] vector x:  1.273008849557522 -0.272787610619469 1.0002212389380531
0.181858407079646 0.8183628318584071

[#2] alpha: -0.21079508863553792
[#2] beta: 0.6169620165866233
[#2] vector x:  0.7929665140501904 0.5735010490761429 1.0406212612908237 -
0.19065973752006998 1.2504484283306296

[#3] alpha: -0.540640071113711
[#3] beta: 4.13488622142942
[#3] vector x:  1.2279026262714465 2.604384340288436 0.9239977208602196 -
2.067555466821925 0.8133484087521211

[#4] alpha: 0.14965872463669921
[#4] beta: 9.233904474686772e-5
[#4] vector x:  0.9928669427638004 1.0001462355982342 1.0172361396633602
0.9989302215202436 0.9901290910427124

[#5] alpha: 0.26535029813869926
[#5] beta: 9.149087504033985e-25
[#5] vector x:  0.9999999999999954 0.999999999999999 0.9999999999999962
1.0000000000000002 0.9999999999999969

x:  0.9999999999999954 0.999999999999999 0.9999999999999962 1.0000000000000002
0.9999999999999969
```

5. Исходный код конвертера файлов .mtx в .csc.

```
import java.io.*;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;

public class MtxToCscConverter {
    public static void main(String[] args) throws IOException {
        if (args == null || args.length < 2) {
            System.out.println("Использование: [входной файл] [выходной файл]");
            System.exit(1);
        }
        File input = new File(args[0]);
        File output = new File(args[1]);
        if (!input.exists() || !input.canRead()) {
            System.out.printf("Файл \"%s\" не существует или нет прав на чтение\n",
input.getAbsolutePath());
            System.exit(2);
        }
        if (output.isDirectory()) {
            System.out.printf("\"%s\" - это директория\n",
output.getAbsolutePath());
            System.exit(3);
        }
        if (output.exists() && output.isFile() && output.canWrite()) {
```

```
            System.out.printf("Файл \"%s\" уже существует. Перезаписать?\n",
output.getAbsolutePath());
            Scanner scanner = new Scanner(System.in);
            String answer = scanner.nextLine();
            if (!answer.equalsIgnoreCase("y") && !answer.equalsIgnoreCase("д"))
System.exit(4);
        }
        int skip = 2;
        String line;
        LinkedList<String> rows = new LinkedList<>();
        LinkedList<String> cols = new LinkedList<>();
        LinkedList<String> values = new LinkedList<>();
        BufferedReader reader = new BufferedReader(new FileReader(input));
        while ((line = reader.readLine()) != null) {
            if (skip-- > 0) {
                if (skip == 1 && !line.startsWith("%%")) {
                    System.out.printf("Файл \"%s\" записан не в формате
MatrixMarket\n", input.getAbsolutePath());
                    System.exit(4);
                }
                continue;
            }
            String[] digits = line.split("\\s+");
            rows.add(digits[0].trim() + " ");
            cols.add(digits[1].trim() + " ");
            values.add(digits[2].trim() + " ");
        }
        reader.close();
        output.delete();
        output.createNewFile();
        FileOutputStream outputStream = new FileOutputStream(output);
        rows.stream().map(String::getBytes).forEach(r -> {
            try {
                outputStream.write(r);
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        outputStream.write("\n".getBytes());
        cols.stream().map(String::getBytes).forEach(c -> {
            try {
                outputStream.write(c);
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        outputStream.write("\n".getBytes());
        values.stream().map(String::getBytes).forEach(v -> {
            try {
                outputStream.write(v);
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        outputStream.close();
```

```
        System.out.println("Конвертация выполнена");
    }
}
```