**РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ**

**Факультет физико-математических и естественных наук**

**Кафедра теории вероятностей и кибербезопасности**

**ОТЧЁТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ №1**

*дисциплина: Параллельное программирование*

Студент: Добычин Дмитрий Романович

Студ. билет № 1032217059

Группа: НПИбд-01-21

**МОСКВА**

2024 г.

# Общая информация:

# Язык программирования: Fortran

**Задание №1:**

У театральной кассы стоят в очереди 2𝑛 человек. Среди них 𝑛 человек имеет монеты только рублевого достоинства, а остальные только монеты по 50 копеек. Билет стоит 50 копеек. Каждый покупатель приобретает по одному билету. В начальный момент в кассе нет денег. Чему равна вероятность того, что ни один покупатель не будет ждать сдачу?

**Описание работы программы:**

Задача состоит в том, чтобы смоделировать очередь из 2𝑛 человек, половина из которых имеет только 1 рубль, а другая половина – только 50 копеек. Необходимо определить вероятность того, что никто не будет ждать сдачу. Для решения задачи можно воспользоваться методом Монте-Карло, перемешивая покупателей случайным образом и проверяя траектории.

1. Заполнение очереди: покупатели с рублями получают значение +1, с 50 копейками – значение -1. В массиве Q первая половина содержит покупателей с рублями, вторая – с полтинниками.
2. Перемешивание очереди: происходит обмен случайных элементов массива с помощью подпрограммы shuffle\_queue.
3. Проверка траектории: суммируются элементы массива слева направо. Если в какой-то момент сумма становится отрицательной, это означает, что покупатель не может получить сдачу и траектория считается неуспешной.
4. Подсчет успешных траекторий: если ни один покупатель не ждал сдачу, траектория считается успешной, и увеличивается счетчик успешных симуляций.
5. Вывод вероятности: после завершения всех симуляций выводится оценка вероятности успешной траектории.

**Реализация на Fortran задания №1:**

В коде 𝑛 = 1000 и проводится 10000 экспериментов.

**program** queue\_simulation

**use** iso\_fortran\_env

**implicit none**

! Параметры программы

**integer**(int64), **parameter** **::** n = 1e3

**integer**(int8), **allocatable** **::** Q(:)

**integer**(int64) **::** i, success\_count, total\_simulations

**real** **::** probability

! Параметры моделирования

total\_simulations = **10000**

! Инициализация динамического массива Q для очереди

**if** (**.not.** allocated(Q)) **allocate**(Q(**2**\*n))

! Основной цикл по симуляциям

success\_count = **0**

**do** i = **1**, total\_simulations

**call** fill\_queue(Q, n)

**call** shuffle\_queue(Q)

**if** (check\_trajectory(Q)) **then**

success\_count = success\_count + **1**

**end if**

**end do**

! Вычисление вероятности как доли успешных симуляций

probability = **real**(success\_count) / **real**(total\_simulations)

**print** '(A, F10.6)', 'Вероятность того, что никто не будет ждать сдачу: ', probability

**contains**

! Подпрограмма для заполнения очереди

**subroutine** fill\_queue(Q, n)

**implicit none**

**integer**(int8), **dimension**(:), **intent**(out) **::** Q

**integer**(int64), **intent**(in) **::** n

**integer**(int64) **::** i

! Заполняем первую половину массива покупателями с рублями, вторую — покупателями с полтинниками

**do** i = **1**, n

Q(i) = **1**\_int8

Q(n + i) = -**1**\_int8

**end do**

**end subroutine** fill\_queue

! Подпрограмма для перемешивания очереди

**subroutine** shuffle\_queue(Q)

**implicit none**

**integer**(int8), **dimension**(:), **intent**(inout) **::** Q

**integer**(int8) **::** temp

**real** **::** u

**integer** **::** i, j, m

m = size(Q) ! Определяем размер массива Q

**do** i = **1**, m

**call** random\_number(u)

j = int(u \* m) + **1**

! Обмен элементов Q(i) и Q(j)

temp = Q(i)

Q(i) = Q(j)

Q(j) = temp

**end do**

**end subroutine** shuffle\_queue

! Функция для проверки траектории

**logical function** check\_trajectory(Q)

**implicit none**

**integer**(int8), **dimension**(:), **intent**(in) **::** Q

**integer**(int64) **::** i, sum

sum = **0** ! Инициализируем сумму

check\_trajectory = .true.

**do** i = **1**, size(Q)

sum = sum + Q(i)

**if** (sum < **0**) **then**

check\_trajectory = .false.

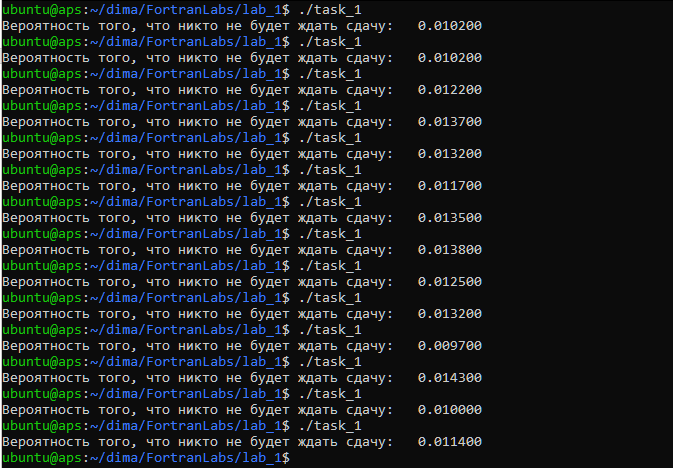
**exit**

**end if**

**end do**

**end function** check\_trajectory

**end program** queue\_simulation

****

**Рис. 1.** Результат работы программы для задания №1.

**Задание №2:**

Брошены две игральные кости. Чему равна вероятность того, что сумма выпавших на них очков равна 8 (событие 𝐴), если известно, что эта сумма есть четное число (событие 𝐵).

**Описание работы программы:**

1. Сгенерировать много пар случайных чисел (симулировать броски костей).
2. Посчитать, сколько раз выполняется событие «сумма четная».
3. Из случаев, когда сумма четная, посчитать, сколько раз сумма равна 8.
4. Рассчитать вероятность по формуле.

**Реализация на Fortran задания №2:**

**program** conditional\_probability

**use** iso\_fortran\_env

**implicit none**

! Параметры

**integer**(int64) **::** num\_trials = **100000**

**integer**(int64) **::** i

**integer**(int64) **::** die1, die2

**integer**(int64) **::** total

**integer**(int64) **::** even\_sum\_count

**integer**(int64) **::** event\_a\_count

**real** **::** probability

! Инициализация счётчиков

even\_sum\_count = **0**

event\_a\_count = **0**

! Генерация случайных бросков двух костей

**do** i = **1**, num\_trials

die1 = int(**6** \* random\_value() + **1**)

die2 = int(**6** \* random\_value() + **1**)

total = die1 + die2

! Проверяем, выполняется ли событие B (четная сумма)

**if** (mod(total, **2**) == **0**) **then**

even\_sum\_count = even\_sum\_count + **1**

! Проверяем выполнение события A (сумма равна 8)

**if** (total == **8**) **then**

event\_a\_count = event\_a\_count + **1**

**end if**

**end if**

**end do**

! Вычисление условной вероятности P(A|B)

probability = **real**(event\_a\_count) / **real**(even\_sum\_count)

! Вывод результата

**print** \*, 'Условная вероятность P(A|B): ', probability

**contains**

! Функция для генерации случайного числа от 0 до 1

**real function** random\_value()

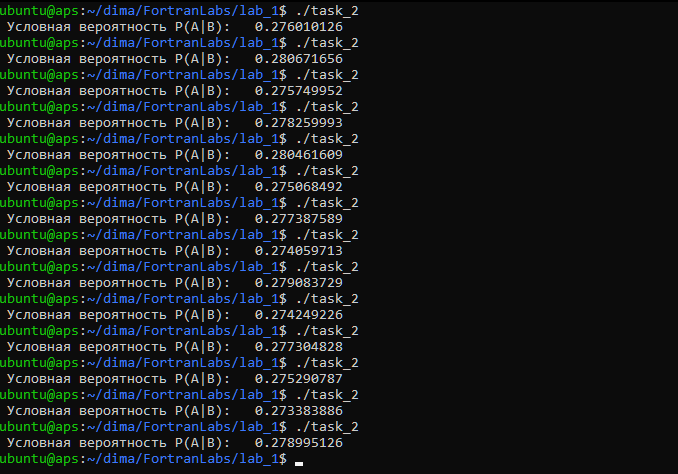
**real** **::** u

**call** random\_number(u)

random\_value = u

**end function** random\_value

**end program** conditional\_probability

****

**Рис. 2.** Результат работы программы для задания №2.

**Задание №3:**

Из колоды карт последовательно вынимают две карты. Найти:

1. Безусловную вероятность того, что вторая карта окажется тузом любой масти. При этом неизвестно какая карта была вынута вначале.
2. Условную вероятность, что вторая карта окажется тузом, если первоначально был вынут туз.

**Реализация на Fortran задания №3:**

**program** card\_probability

**implicit none**

**integer**, **parameter** **::** trials = **1000000**

**integer** **::** i, ace\_count\_A = **0**, ace\_count\_AB = **0**

**integer** **::** first\_card, second\_card

**character**(len=**6**), **dimension**(**36**) **::** deck, shuffled\_deck

! Определим колоду карт (36 карт)

deck = [ & ! Колода состоит из 36 карт, каждая карта имеет символическую длину 6 (чтобы учесть масть и номинал)

"T♣ ", "K♣ ", "D♣ ", "V♣ ", "6♣ ", "7♣ ", "8♣ ", "9♣ ", "10♣ ", & ! Трефы

"T♠ ", "K♠ ", "D♠ ", "V♠ ", "6♠ ", "7♠ ", "8♠ ", "9♠ ", "10♠ ", & ! Пики

"T♦ ", "K♦ ", "D♦ ", "V♦ ", "6♦ ", "7♦ ", "8♦ ", "9♦ ", "10♦ ", & ! Бубны

"T♥ ", "K♥ ", "D♥ ", "V♥ ", "6♥ ", "7♥ ", "8♥ ", "9♥ ", "10♥ " ] ! Червы

! Повторим испытание trials раз

**do** i = **1**, trials

shuffled\_deck = deck

**call** shuffle\_deck(shuffled\_deck)

! Выберем две карты

first\_card = **1**

second\_card = **2**

! Проверяем, является ли вторая карта тузом

**if** (index(shuffled\_deck(second\_card), "T") == **1**) **then**

ace\_count\_A = ace\_count\_A + **1**

**end if**

! Проверяем условную вероятность: если первая карта — туз и вторая карта тоже туз

**if** (index(shuffled\_deck(first\_card), "T") == **1** **.and.** index(shuffled\_deck(second\_card), "T") == **1**) **then**

ace\_count\_AB = ace\_count\_AB + **1**

**end if**

**end do**

! Выводим результаты

**print** '(A, F10.6)', "Безусловная вероятность P(A) того, что вторая карта туз: ", **real**(ace\_count\_A) / **real**(trials)

**print** '(A, F10.6)', "Условная вероятность P(A|B) того, что вторая карта туз, если первая туз: ", **real**(ace\_count\_AB) / **real**(ace\_count\_A)

**contains**

! Процедура перемешивания колоды карт (алгоритм Fisher-Yates)

**subroutine** shuffle\_deck(deck)

**implicit none**

**character**(len=\*), **dimension**(:), **intent**(inout) **::** deck

**character**(len=**6**) **::** temp

**real** **::** r

**integer** **::** i, j, n

n = size(deck)

! Fisher-Yates перемешивание

**do** i = n, **2**, -**1**

**call** random\_number(r)

j = int(r \* i) + **1**

temp = deck(i)

deck(i) = deck(j)

deck(j) = temp

**end do**

**Рис. 3.** Результат работы программы для задания №3.

**Задание №4:**

Рассмотрим следующую простую программу (программа дана внутри файла лабораторной работы №1). В данной программе для нахождения скалярного произведения двух массивов используется функция sum в комбинации с умножением \* и, отдельно, функция dot\_product.

Скомпилируйте программу используя следующие флаги компилятора:

* -O3 — третий уровень оптимизации;
* -fopt-info-vec-all — показывать всю информацию о попутках векторизировать циклы и другие инструкции;

Ответьте на следующие вопросы:

1. Какие выводы можно сделать из сообщений компилятора?
2. Что такое векторизация? Что такое SIMD инструкции?
3. Замените нахождения скалярного произведения на суммирование в цикле и проверьте, сумел ли компилятор (а точнее оптимизатор) осуществить векторизацию?
4. Замените статические массивы на динамические и выясните будут ли векторизироваться действия с ними, ориентируйтесь на сообщения компилятора.

**Реализация на Fortran задания №4:**

**Пункт №1:**

**program** vectorize

**implicit none**

**real**, **dimension**(**7**) **::** a, b

! Инициализация массивов

a = [**2.0**, **3.0**, **5.0**, **7.0**, **11.0**, **13.0**, **17.0**]

b = [**4.0**, **6.0**, **8.0**, **10.0**, **12.0**, **14.0**, **16.0**]

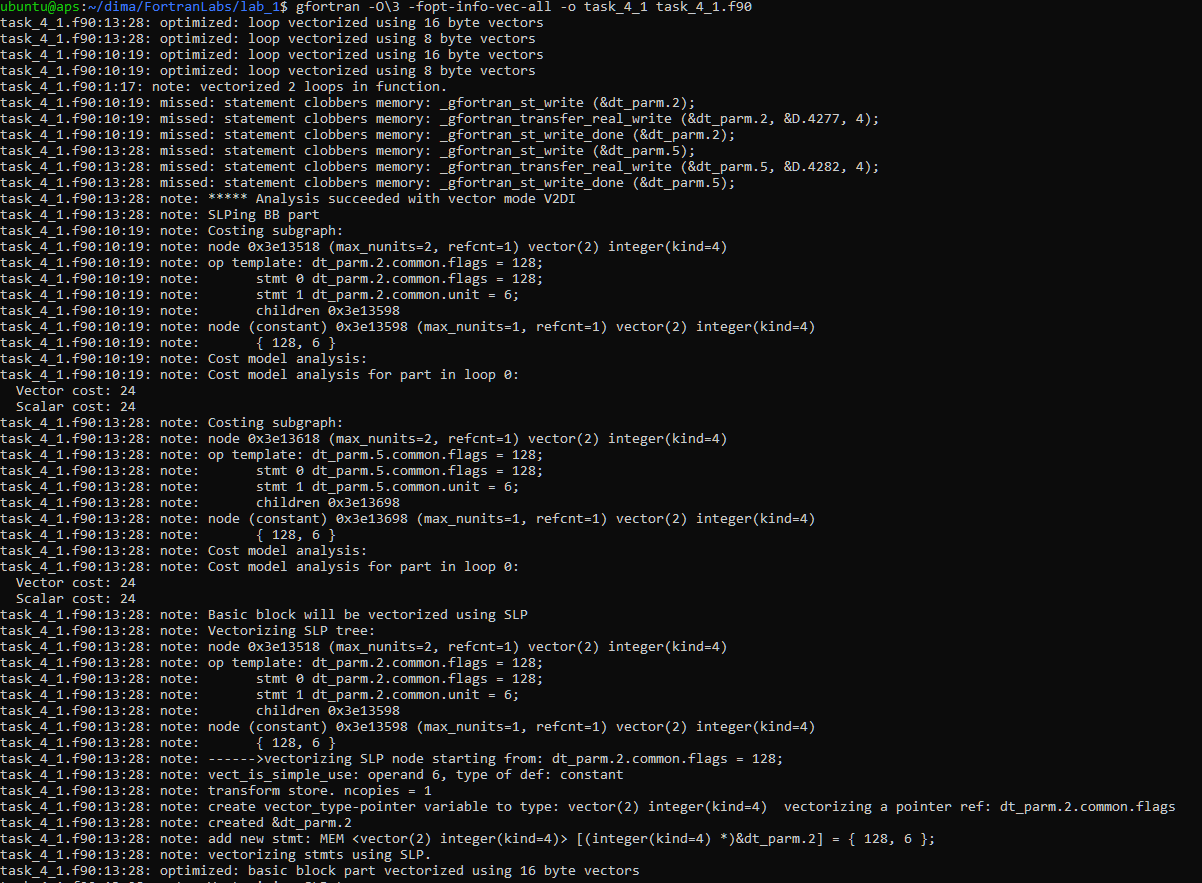
! Вывод суммы произведения элементов массивов

**print** \*, sum(a\*b)

! Вывод результата через встроенную функцию dot\_product

**print** \*, dot\_product(a, b)

**end program** vectorize

****

**Рис. 4.1.** Результат работы программы для пункта №1.

**Успешная векторизация циклов:** сообщения типа "optimized: loop vectorized using 16 byte vectors" указывают на то, что компилятор успешно применил векторизацию. Это означает, что компилятор преобразовал операции с массивами в инструкции SIMD (Single Instruction, Multiple Data), которые могут одновременно выполнять одну и ту же операцию над несколькими элементами данных. Векторизация приводит к улучшению производительности программы, особенно при работе с большими данными.

**Применение техники SLP (Superword Level Parallelism):** сообщения о SLP говорят о том, что компилятор применил технику параллелизма на уровне суперслов для векторизации базовых блоков программы. Эта техника объединяет несколько независимых операций с данными в одну SIMD-инструкцию, что еще больше ускоряет выполнение программы.

**Пропущенные возможности векторизации (I/O операции):** Сообщения вида "missed: statement clobbers memory" указывают на то, что компилятор не смог векторизовать некоторые участки кода. Обычно это связано с операциями ввода-вывода (print), которые работают с памятью и могут иметь побочные эффекты, делающие их сложными для векторизации. Это нормальное поведение, так как операции I/O не векторизуются из-за их природы (последовательность важна).

**Соответствие затрат на векторные и скалярные операции:** Сообщения о сравнении vector cost и scalar cost показывают, что компилятор оценивает и сравнивает затраты на выполнение операций как в векторной, так и в скалярной форме. Если стоимость векторизации и скалярного кода примерно одинаковая, это означает, что выигрыш от векторизации может быть небольшим.

**SIMD инструкции с различной шириной (8 и 16 байт):** векторизация была выполнена как с использованием 16-байтных, так и 8-байтных векторов. Это зависит от возможностей процессора и типов данных. Компилятор старается максимально эффективно использовать доступные ресурсы процессора, что свидетельствует о том, что код будет работать оптимально на различных архитектурах.

**Ограничения векторизации в определенных случаях:** когда компилятор не может гарантировать, что инструкции не изменяют память непредсказуемым образом (например, через побочные эффекты или сложные зависимости), векторизация не применяется. Это видно по сообщениям о пропущенных возможностях векторизации (missed: statement clobbers memory).

**Пункт №2:**

**Векторизация** — это процесс преобразования обычных (скалярных) операций с данными в такие, которые могут одновременно обрабатывать несколько элементов данных за одну инструкцию. В контексте программирования, это оптимизация, которая позволяет компилятору или программисту применять SIMD-инструкции для выполнения одной и той же операции сразу на нескольких элементах массива (или другого набора данных).

Например, если у вас есть массив чисел и нужно сложить все его элементы, без векторизации это происходит последовательно: сначала складываются первые два элемента, затем результат складывается со следующим элементом и так далее. В случае векторизации несколько операций сложения могут выполняться параллельно, что значительно ускоряет вычисления на современных процессорах.

Основная цель векторизации — повышение производительности за счет использования возможностей современных процессоров выполнять несколько операций одновременно.

**SIMD (Single Instruction, Multiple Data)** — это тип инструкций, который позволяет процессору выполнять одну и ту же операцию над несколькими данными одновременно. Это основа векторизации.

В обычных (скалярных) операциях процессор выполняет одну операцию (например, сложение или умножение) над одним или двумя операндами. В случае SIMD-инструкций одна операция может применяться к нескольким элементам данных, что ускоряет вычисления.

**Пункт №3:**

**program** vectorize

**implicit none**

**real**, **dimension**(**7**) **::** a, b

**real** **::** **result**

**integer** **::** i

! Инициализация массивов

a = [**2.0**, **3.0**, **5.0**, **7.0**, **11.0**, **13.0**, **17.0**]

b = [**4.0**, **6.0**, **8.0**, **10.0**, **12.0**, **14.0**, **16.0**]

! Вычисление скалярного произведения через цикл

**result** = **0.0**

**do** i = **1**, **7**

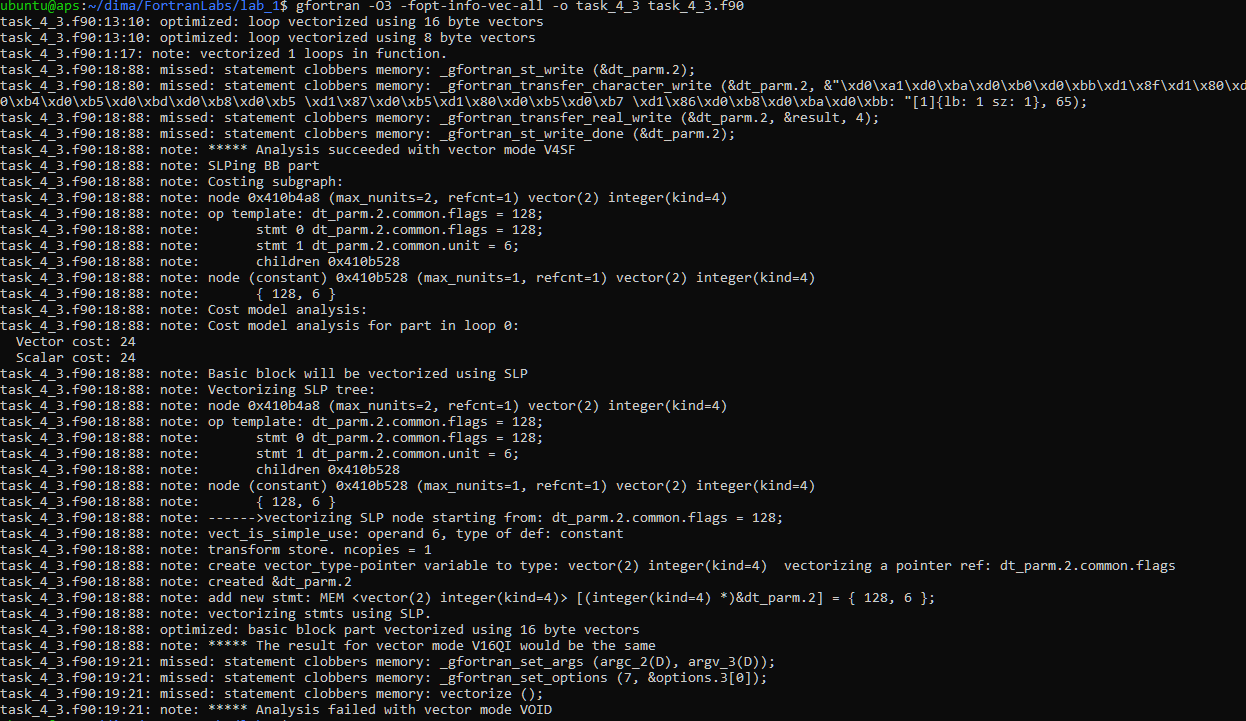
**result** = **result** + a(i) \* b(i)

**end do**

! Вывод результата

**print** \*, "Скалярное произведение через цикл: ", **result**

**end program** vectorize

****

**Рис. 4.2.** Результат работы программы для пункта №3.

Компилятор успешно векторизовал основной цикл, используемый для суммирования произведений элементов массивов a и b. Это видно из сообщений:

* loop vectorized using 16 byte vectors
* loop vectorized using 8 byte vectors

Это означает, что компилятор применил инструкции SIMD (Single Instruction, Multiple Data), чтобы обработать несколько элементов массива одновременно, что ускоряет выполнение программы.

Сообщения компилятора, такие как SLPing BB part и Vectorizing SLP tree, указывают на успешную векторизацию основного цикла с использованием техники SLP.

Компилятор не смог векторизовать операции вывода, например:

* statement clobbers memory: \_gfortran\_st\_write

Это связано с тем, что ввод-вывод не подлежит векторизации, так как такие операции не являются "параллелизуемыми" по своей природе.

Таким образом, компилятор сумел векторизовать цикл для нахождения скалярного произведения при его реализации через цикл.

**Пункт №4:**

**program** vectorize\_dynamic

**implicit none**

**integer** **::** n

**real**, **allocatable** **::** a(:), b(:)

**real** **::** **result**, dot\_result

**integer** **::** i

! Задайте размерность массива

n = **7**

**allocate**(a(n), b(n))

! Инициализация массивов

a = [**2.0**, **3.0**, **5.0**, **7.0**, **11.0**, **13.0**, **17.0**]

b = [**4.0**, **6.0**, **8.0**, **10.0**, **12.0**, **14.0**, **16.0**]

! Скалярное произведение с использованием суммирования в цикле

**result** = **0.0**

**do** i = **1**, n

**result** = **result** + a(i) \* b(i)

**end do**

**print** \*, "Скалярное произведение через цикл:", **result**

! Скалярное произведение с использованием dot\_product

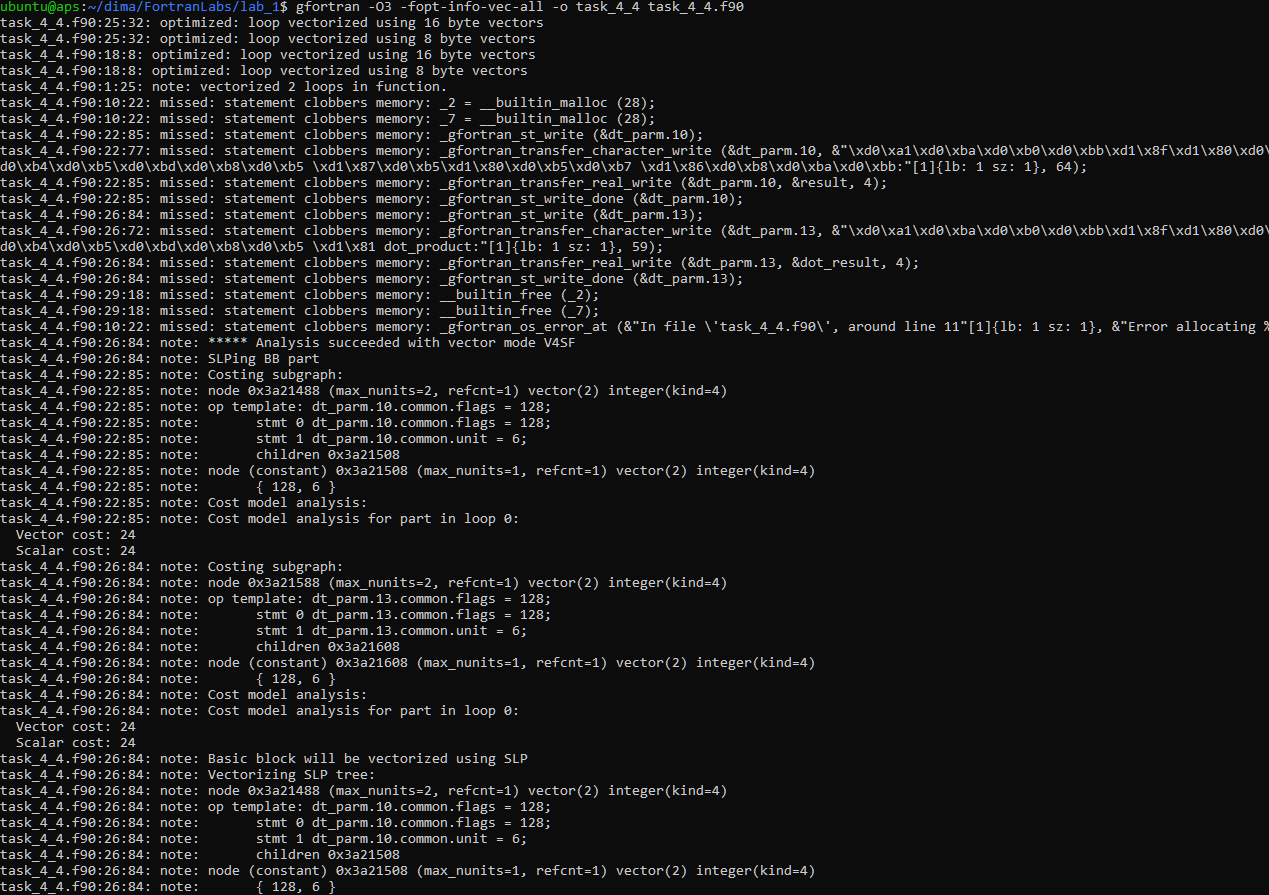
dot\_result = dot\_product(a, b)

**print** \*, "Скалярное произведение с dot\_product:", dot\_result

! Освобождение памяти

**deallocate**(a, b)

**end program** vectorize\_dynamic

****

**Рис. 4.3.** Результат работы программы для пункта №4.

Векторизация успешна:

optimized: loop vectorized using 8 byte vectors

optimized: loop vectorized using 16 byte vectors

Это указывает на то, что некоторые циклы были успешно векторизированы.

Неудачная векторизация:

missed: statement clobbers memory

Эти сообщения показывают, что компилятор не смог векторизировать некоторые операции из-за потенциального конфликта доступа к памяти (clobbering). Это может происходить, когда компилятор не может гарантировать, что записи и чтения из массивов не пересекаются.

Когда вы используете динамические массивы, компилятор может быть менее оптимистичен относительно возможности векторизации. Это связано с тем, что динамическое выделение памяти затрудняет анализ доступа к данным.