**Отчет о выполнении практического задания по предмету**

**«Прикладное программирование в задачах науки и техники»**

**на тему**

**«Разработка параллельной программы с использованием технологии OpenMP»**

Выполнил:

Федосеев Артём Леонидович

Аспирант первого года

ИПМ РАН

Москва, 2023

# Оглавление

## Постановка задачи

1) Требуется разработать две версии параллельного исполнения предложенной программы с использованием технологии OpenMP
2) Исследовать правильность реализованной параллельной версии программы при помощи средств анализа параллельных программ пакета InspXe
3) Провести сравнительный анализ реализованных версий программ с использованием различных компиляторов (gcc, icc, pgcc, clang) и различных уровней оптимизации (O0, 01, 02, 03)
4) Провести сравнение скорости работы предложенных версий программы на различных объемах данных с различным числом нитей, используемых при исполнении, с использованием различных компиляторов (gcc, icc, pgcc, clang)
5) Выбрать оптимальные параметры для исполнения программы и определить основные причины недостаточной масштабируемости программы

### Использованные компиляторы

1) gcc (GCC) 10.2.0 Copyright (C) 2020 Free Software Foundation, Inc.
2) icc (ICC) 17.0.0 20160721 Copyright (C) 1985-2016 Intel Corporation.
   All rights reserved.
3) pgcc не запустился со следующей ошибкой:
   pgcc: /lib64/libc.so.6: version `GLIBC_2.14' not found (required by pgcc)
4) clang не запустился со следующей ошибкой, обозначающей отсутствие поддержки OpenMP.
   var11_for.c:4:10: fatal error: 'omp.h' file not found

Поэтому анализ производился при сравнении двух компиляторов, gcc и icc.

## Описание программы

### Базовый алгоритм

В качестве базового алгоритма мне был предложен алгоритм итеративного процесса для двумерной сетки с ранней, в случае сходимости, остановкой, в котором во время расчета новой итерации в памяти хранится целиком состояние сетки на прошлой итерации. Благодаря этому основной расчетный цикл хорошо параллелизуется, так как зависимости чтения-запись разведены на два разных этапа.

Он состоит из четырех базовых функций: *init, relax, resid, verify.* В функции *init* происходит инициализация массива, над которым будет производиться основная вычислительная работа, в функции *relax* происходит выполнение основной вычислительной работы, при этом функция запускается в цикле до сходимости задачи. Внутри функции *resid* вычисляется максимум поэлементной разницы между итерациями, необходимая для ранней остановки процесса. В ней же происходит перезапись старой итерации результатами новой. Функция *verify* производит вычисление контрольной суммы по массиву данных, после завершения процесса итерации.

### Общие модификации алгоритма

Алгоритм почти не модифицировался, не считая перестановки порядка обхода элементов массива, чтобы лучше утилизировать кеш.

Единственное примечательное изменение — в процессе паралеллизации было обнаружно, что компиляторы на K10 плохо поддерживают прагму OpenMP *reduction*, что для *omp for*, что для *omp taskloop* (а в случае icc в целом нет поддержки *omp taskloop reduction* и *task_reduction*, хотя это часть стандарта OpenMP 3.0), поэтому был заведен массив максимальных eps_errors, который заполняется параллельно в *resid*, по которому потом отдельный поток выполняет редукцию.

Во всех случаях параллелизация производилась только основных вычислительных функций, *relax* и *resid*, как вносящих основной вклад во время работы программы.

Также, для замеров времени везде использовалась *omp_get_wtime().*

### Версия программы с omp for

Была произведена параллелизация основных циклов с помощью прагмы:

*#pragma omp for schedule(static)*

Распределение было выбрано статичным, так как нагрузки на всех итерациях цикла по строкам матриц примерно равные.

Директива *collapse(N)* не использовалась, так как это приводило к крашам программ скомпилированных с icc с оптимизацией -O1 и ниже.

Инициализация параллельной секции *omp parallel,* общей для relax и resid, происходит на каждой новой итерации алгоритма, что не должно привести к большим накладным расходам.

### Версия программы с omp task

Для получения большей практики использовались как директива *omp taskloop*, так и *omp taskgroup* с явной генерацией *task* внутри региона.

Генерация параллельной области производится один раз, на весь расчетный цикл, с помощью в меру стандартной последовательности прагм *omp parallel* + *omp master* (можно single).

Использовалась прагма omp master, т. к. по окончанию omp parallel и так есть неявный барьер.

## Экспериментальная оценка реализованных версий программы

Для автоматизации компиляции, запуска программ с разными параметрами, сбора статистики, построения графиков и прочих задач, были использованы скрипты на языке Python, основанные на скрипте, разработанном одногруппником Алексеем Поповым. Также были дописаны некоторые скрипты на Bash, для упрощения перекомпиляции icc версий. Всё это идет приложением к отчету.

Все полученные цифры – результат усреднения нескольких (обычно 5) запусков. Конкретные значения также приложены к отчету отдельными файлами.

### Проверка корректности написанной программы

Для проверки правильности реализованных версий программы была проведена их валидация при помощи компилятора *icc* и инспектора параллельных программ от компании *Intel – InspXe*.

Примеры использованных команд и ключей:

*inspxe-cl -collect=ti3* и *inspxe-cl -collect=mi3*

В результате проверки были выявлены подозрительные места,

которые были дополнительно проверены и обработаны.

**Сравнение различных компиляторов и уровней оптимизации**

В качестве размера стороны матриц взято значение в 4098, второе по размеру значение среди исследованных параметров.

Результаты приложены на Графике 1 и Графике 2 соответственно.

Так как компиляторов всего два, нагляднее демонстрировать в виде нескольких 2D графиков.

Какие выводы можно сделать на основе полученных данных:

1. В icc 17.0.0 поддержка task-параллизма хуже, чем в gcc 10.2.0.
   Это также было видно по тому, что некоторые директивы в icc вообще не поддерживались (*taskloop reduction*, падения с применением *collapse(N)* ), в то время как в gcc не было проблем;
2. При этом более понятный for-параллелизм *icc* использует лучше — начиная с O2 «родной» компилятор для используемого на кластере железа порождает более быструю программу, при условии что это еще версия компилятора на несколько лет более старого чем установленный gcc;
3. gcc на O1 оптимизирует агрессивнее, чем icc, начиная с O1 и дальше у gcc почти нет разницы;
4. icc, в случае O0, производит самые долгоработающие программы.

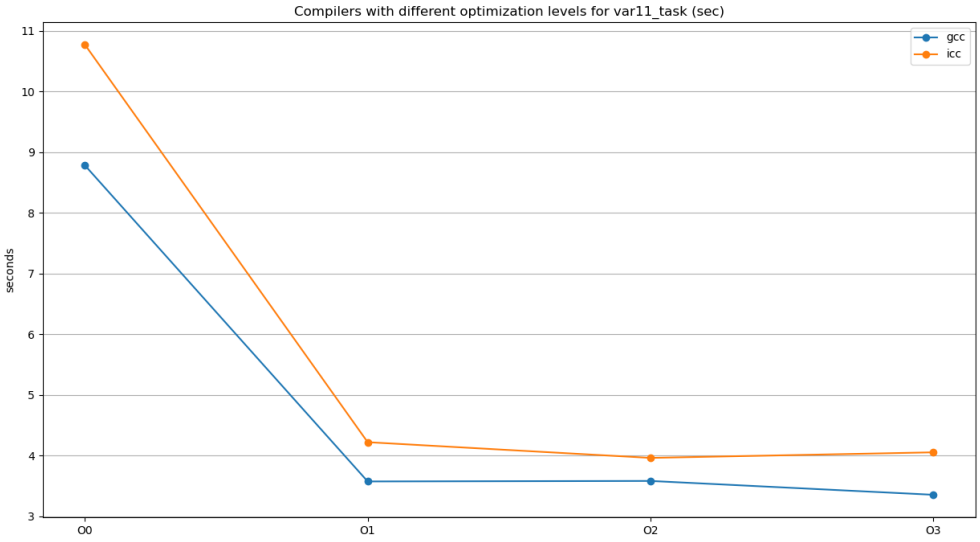



*График 1: Сравнение уровней оптимизаций для var11_task*

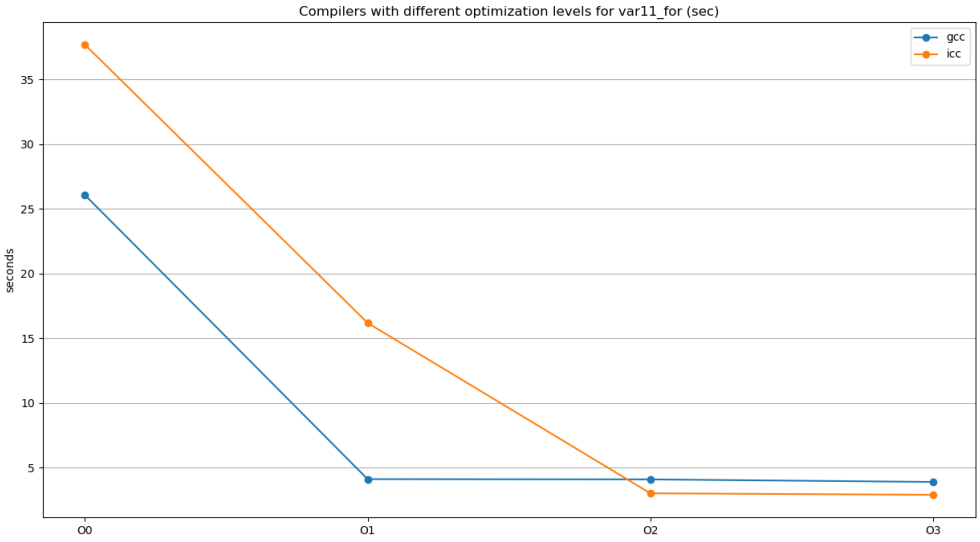



*График 2: Сравнение уровней оптимизаций для var11_for*

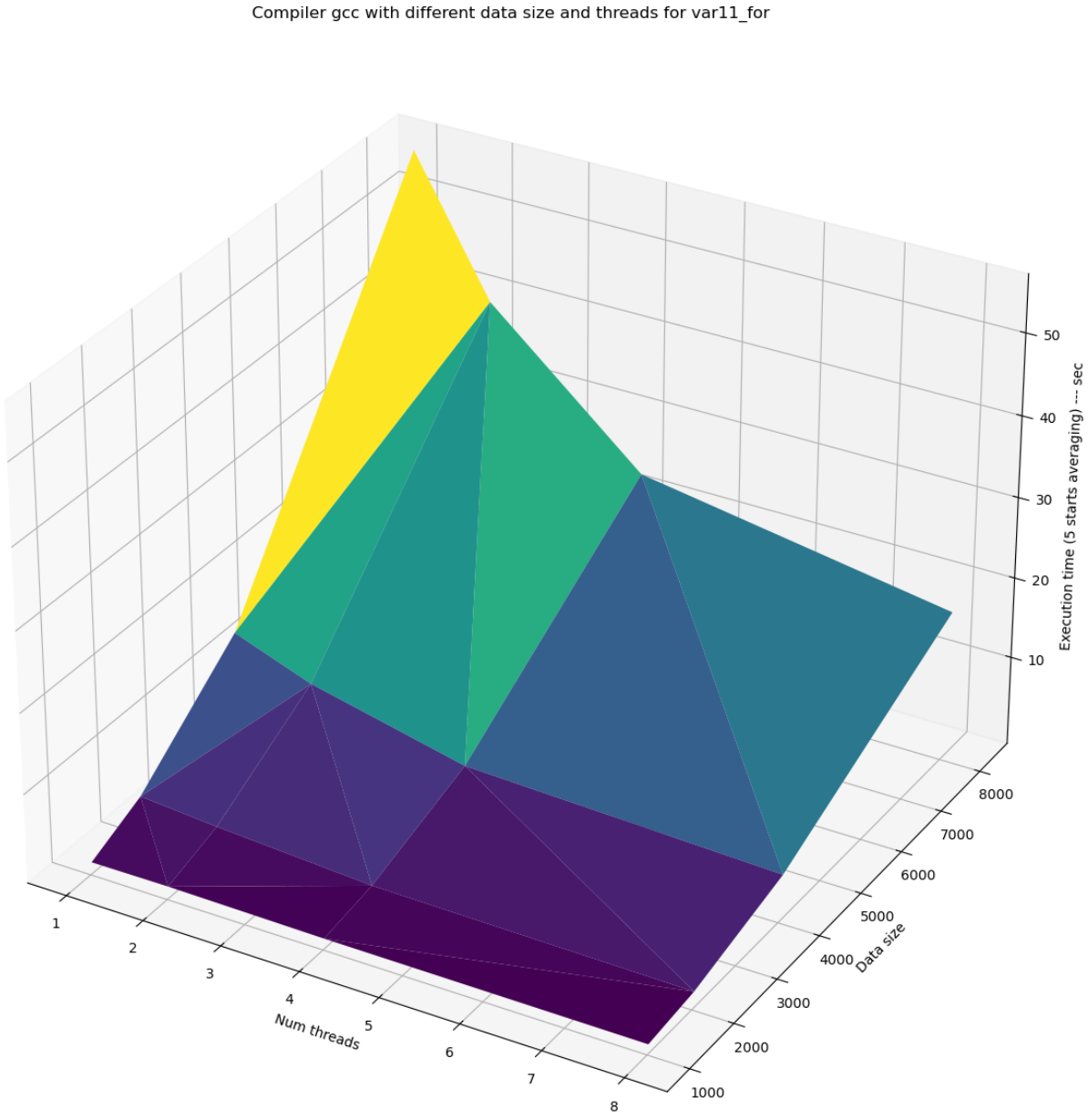
**Сравнение различных компиляторов, размеров данных и количестве потоков**

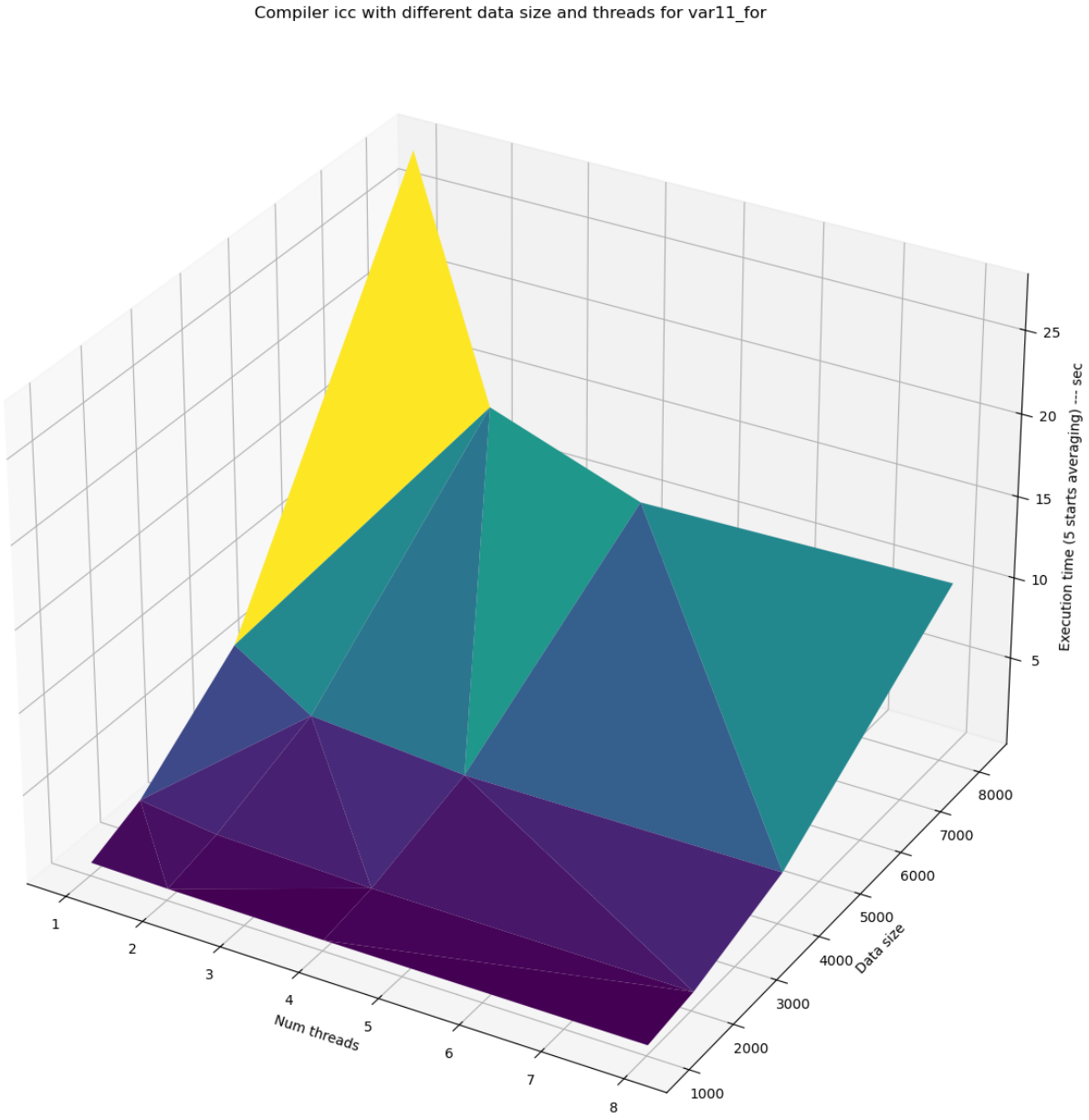
В качестве уровня оптимизации был выбран O3, как самый быстрый и при этом сохраняющий корректность работы программ.

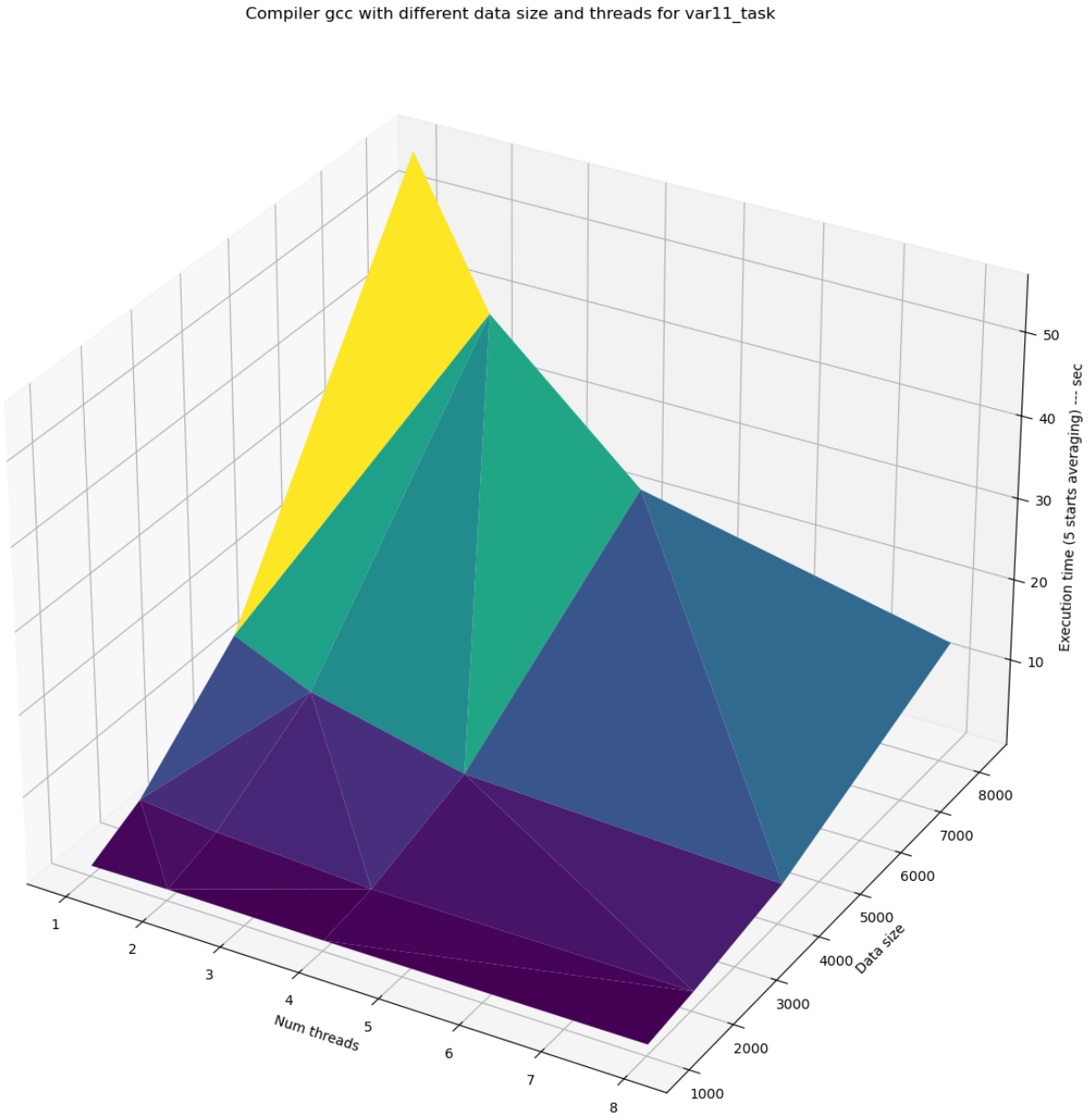
Результаты в виде 3D графиков можно увидеть далее.

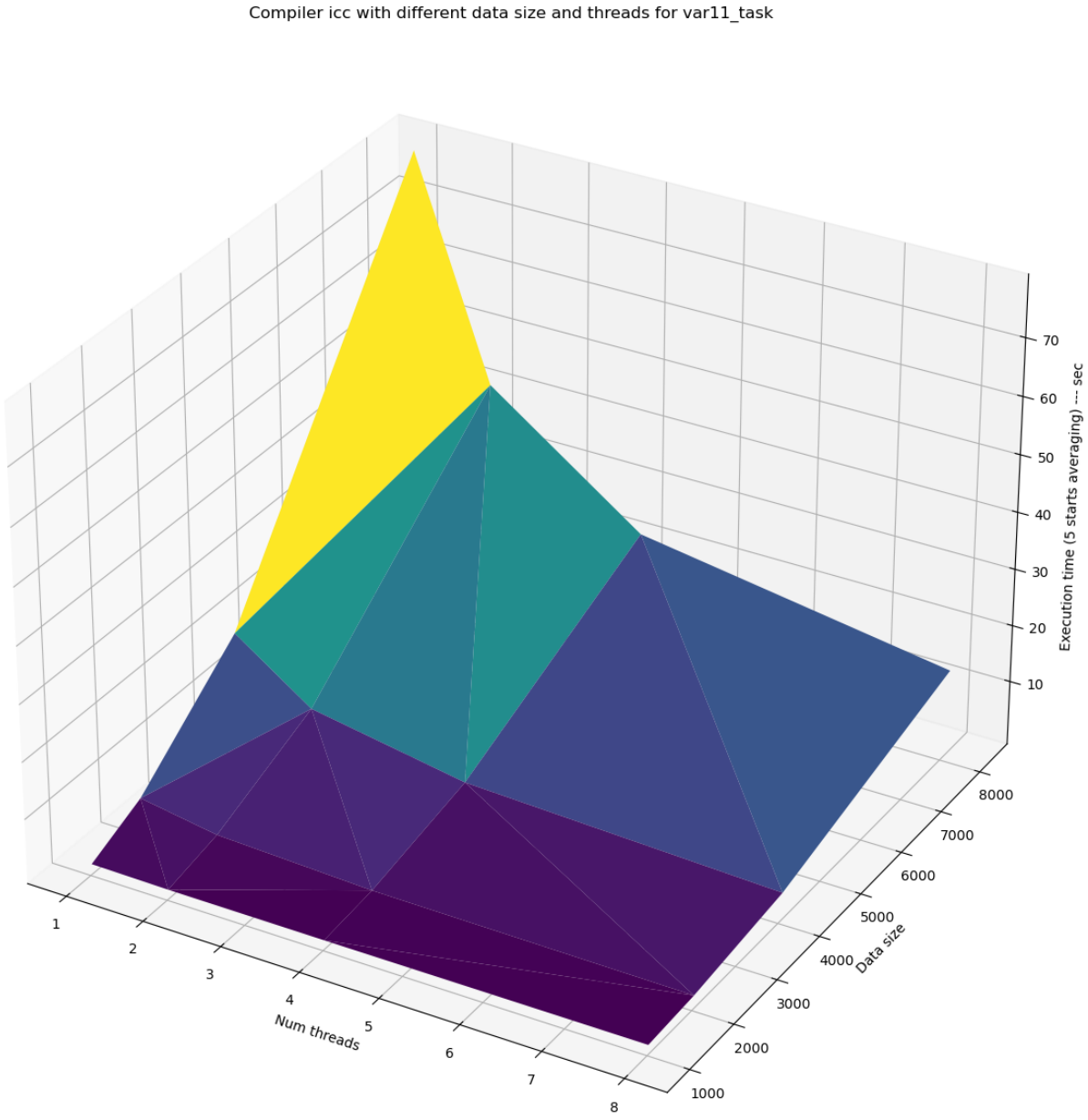
Выводы, которые можно сделать на основе полученных данных:

1. *icc,* будучи «родным» для железа компилятором, лучше справляется при большом числе потоков, как в *for* случае, так и *task*. При этом для маленького числа потоков он наоборот, хуже.
2. Программа хорошо масштабируется — при росте числа потоков на хоть сколько-то значимом объеме данных, не помещающихся в кеш (размер матрицы - 8194), продолжает происходить ускорение, а в случае task — еще и достаточно близкое к линейному. Иначе говоря - предел масштабируемости программы определить для заданных параметров не удалось.


Compiler gcc with different data size and threads for var11_for
Num threads
Data size
Execution time (5 starts averaging) --- sec
1
2
3
4
5
6
7
8
1000
2000
3000
4000
5000
6000
7000
8000
10
20
30
40
50


Compiler icc with different data size and threads for var11_for
Num threads
Data size
Execution time (5 starts averaging) --- sec
1
2
3
4
5
6
7
8
1000
2000
3000
4000
5000
6000
7000
8000
5
10
15
20
25


Compiler gcc with different data size and threads for var11_task
Num threads
Data size
Execution time (5 starts averaging) --- sec
1
2
3
4
5
6
7
8
1000
2000
3000
4000
5000
6000
7000
8000
10
20
30
40
50


Compiler icc with different data size and threads for var11_task
Num threads
Data size
Execution time (5 starts averaging) --- sec
1
2
3
4
5
6
7
8
1000
2000
3000
4000
5000
6000
7000
8000
10
20
30
40
50
60
70