

Министерство образования Республики Беларусь  
Учреждение образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №3

По дисциплине «Архитектуры вычислительных систем»  
По теме «Программирование многоядерных архитектур»

Выполнил:  
студент гр. 953501  
Корневский С. А.

Проверил:  
старший преподаватель  
Шиманский В.В.

Минск 2021

## Содержание

1. Цель работы .....	3
2. Постановка задачи .....	4
3. Краткие теоретические сведения .....	5
3.1. Многопоточность и мультизадачность .....	5
3.2. Параллельное программирование .....	7
3.3. Параллельные архитектуры .....	7
3.4. Разработки Intel .....	8
3.4.1 Hyper-threading .....	8
3.4.2 Parallel Studio XE 2019 .....	10
3.5 OpenMP .....	12
3.6. Разработки Intel .....	14
Intel Parallel Advisor .....	15
Intel Parallel Composer .....	16
Intel Parallel Inspector .....	17
4. Демонстрация работы программы .....	22
5. Графики .....	25
Вывод .....	27
Приложение 1. Исходный код программы .....	28

## **1. Цель работы**

Использование интерфейса OpenMP для программирования многопоточного приложения, выполняющего подсчет скалярного произведения двух векторов. Провести сравнение быстродействия многопоточного и однопоточного алгоритмов.

## **2. Постановка задачи**

Написать программу, реализующую нахождение скалярного произведения двух векторов. Получение числа координат вектора должно осуществляться с клавиатуры. Получить время выполнения однопоточного и многопоточного (4 потока) алгоритмов. Для реализации использовать интерфейс OpenMP.

### 3. Краткие теоретические сведения

#### 3.1. Многопоточность и мультизадачность

Под мультизадачностью подразумевают способность компьютера выполнять несколько задач одновременно. На самом деле процессор некоторое время выполняет один командный поток, затем быстро переключается на второй и выполняет его, переключается на третий и т.д. При этом при каждом переключении сохраняется контекст прерываемого потока, так что потом процессор сможет "безболезненно" продолжить выполнение прерванного потока команд. Благодаря высокому быстродействию создается иллюзия того, что все задачи выполняются одновременно (параллельно).

Сутью многопоточности является квазимногозадачность на уровне одного исполняемого процесса, то есть все потоки выполняются в адресном пространстве процесса. Кроме этого, все потоки процесса имеют не только общее адресное пространство, но и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

Многопоточность (как доктрину программирования) не следует путать ни с многозадачностью, ни с многопроцессорностью, несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность.

К достоинствам многопоточной реализации той или иной системы перед многозадачной можно отнести следующее:

- Упрощение программы в некоторых случаях за счёт использования общего адресного пространства.
- Меньшие относительно процесса временные затраты на создание потока.

К достоинствам многопоточной реализации той или иной системы перед однопоточной можно отнести следующее:

- Упрощение программы в некоторых случаях, за счёт вынесения механизмов чередования выполнения различных слабо взаимосвязанных подзадач, требующих одновременного выполнения, в отдельную подсистему многопоточности.
- Повышение производительности процесса за счёт распараллеливания процессорных вычислений и операций ввода-вывода.

В случае, если потоки выполнения требуют относительно сложного взаимодействия друг с другом, возможно проявление проблем многозадачности, таких как взаимные блокировки.

На обычном процессоре управление потоками осуществляется операционной системой. Поток исполняется до тех пор, пока не произойдёт аппаратное прерывание, системный вызов или пока не истечёт отведённое для него операционной системой время. После этого процессор переключается на код операционной системы, который сохраняет состояние потока (его контекст) или переключается на состояние другого потока, которому тоже выделяется время на исполнение. При такой многопоточности достаточно большое количество тактов процессора тратится на код операционной системы, переключающий контексты. Если поддержку потоков реализовать аппаратно, то процессор сам сможет переключаться между потоками, а в идеальном случае - выполнять несколько потоков одновременно за каждый такт. Для операционной системы и пользователя один такой физический процессор будет виден как несколько логических процессоров.

Различают две формы многопоточности, которые могут быть реализованы в процессорах аппаратно:

- Временная многопоточность (англ. Temporal multithreading)
- Одновременная многопоточность (англ. Simultaneous multithreading)

### 3.2. Параллельное программирование

Параллельное программирование применяется тогда, когда для последовательной программы требуется уменьшить время ее выполнения, или когда последовательная программа, в виду большого объема данных, перестает помещаться в память одного компьютера.

Разработка параллельных программ (ПП) состоит из трех основных этапов:

- Декомпозиция задачи на подзадачи. Идеально, чтобы эти подзадачи работали независимо друг от друга (принцип локальности данных). Обмен данными между подзадачами является дорогой операцией, особенно, если это обмен по сети.
- Распределение задачи по процессорам (виртуальным процессорам). В некоторых случаях решение этого вопроса можно оставить на усмотрение среды выполнения ПП.
- Написание программы с использованием какой-либо параллельной библиотеки.

### 3.3. Параллельные архитектуры

В массе своей все вычислительные комплексы и компьютеры делятся на три группы:

- Системы с распределенной памятью. Каждый процессор имеет свою память и не может напрямую получать доступ к памяти другого процессора. Программист в явном виде должен задать всю систему коммуникации (Передача сообщений – Message Passing). Библиотеки: MPI, PVM, Shmem (Cray only).



Рис. Системы с распределенной памятью

- Системы с общей (разделяемой) памятью. Процессор может напрямую обращаться в память другого процессора. Процессоры могут сидеть на одной шине (SMP). Подходы к разработке ПО: Threads, директивы компилятора (OpenMP), механизм передачи сообщения.

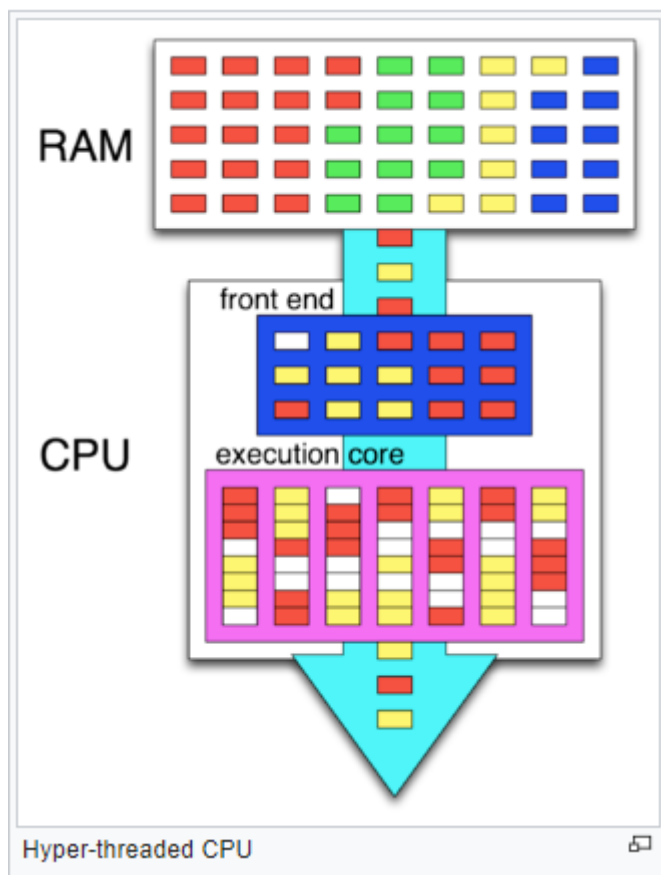


Рис. Система с общей памятью

- Комбинированные системы. В кластерах могут объединяться компьютеры различной конфигурации.

### 3.4. Разработки Intel

#### 3.4.1 Hyper-threading



Hyper-threading (англ. hyper-threading — гиперпоточность, официальное название — hyper-threading technology, НТТ или НТ) — технология, разработанная компанией Intel для процессоров на микроархитектуре NetBurst. НТТ реализует идею «одновременной мультипоточности» (англ. simultaneous multithreading, SMT). НТТ является развитием технологии суперпоточности (англ. super-threading), появившейся в процессорах Intel Xeon в феврале 2002 и в ноябре 2002 добавленной в процессоры Pentium 4. После включения НТТ один физический процессор (одно физическое ядро) определяется операционной системой как два отдельных процессора (два логических ядра).

При определённых рабочих нагрузках использование НТТ позволяет увеличить производительность процессора. Суть технологии: передача «полезной работы» (англ. useful work) бездействующим исполнительным устройствам (англ. execution units).



НТТ не реализована в процессорах серии Core 2 («Core 2 Duo», «Core 2 Quad»).

В процессорах Core i3, Core i7 и некоторых Core i5 была реализована сходная по своим принципам технология, сохранившая название hyper-threading. При включении технологии каждое физическое ядро процессора определяется операционной системой как два логических ядра.

Процессор, поддерживающий технологию hyper-threading:

- может хранить состояние сразу двух потоков;
- содержит по одному набору регистров и по одному контроллеру прерываний (APIC) на каждый логический процессор.

Для операционной системы это выглядит как наличие двух логических процессоров (англ. logical processor). У каждого логического процессора имеется свой набор регистров и контроллер прерываний (APIC). Остальные элементы физического процессора являются общими для всех логических процессоров.

Пример. Физический процессор выполняет поток команд первого логического процессора. Выполнение потока команд приостанавливается по одной из следующих причин:

- произошёл промах при обращении к кэшу процессора;
- выполнено неверное предсказание ветвления;
- ожидается результат предыдущей инструкции.

Физический процессор не будет бездействовать, а передаст управление потоку команд второго логического процессора. Таким образом, пока один логический процессор ожидает, например, данные из памяти, вычислительные ресурсы физического процессора будут использоваться вторым логическим процессором.

Преимуществами НТТ считаются:

- возможность запуска нескольких потоков одновременно (многопоточный код);
- уменьшение времени отклика;
- увеличение числа пользователей, обслуживаемых сервером.
- По утверждениям компании Intel после реализации НТТ в Pentium 4 и Xeon 2001-2002 года:
- площадь кристалла и энергопотребление в первой реализации увеличилась менее чем на 5%[5][6];
- в некоторых задачах производительность увеличилась на 15—30%

- прибавка к скорости составила 30%[8] по сравнению с аналогичными процессорами Pentium 4, не поддерживающими НТТ;

Прибавка к производительности изменяется от приложения к приложению. Скорость выполнения некоторых программ может даже уменьшиться. Это, в первую очередь, связано с «системой повторения» (англ. replay) процессоров Pentium 4, занимающей необходимые вычислительные ресурсы, отчего и начинают «голодать» другие потоки.

### **3.4.2 Parallel Studio XE 2019**

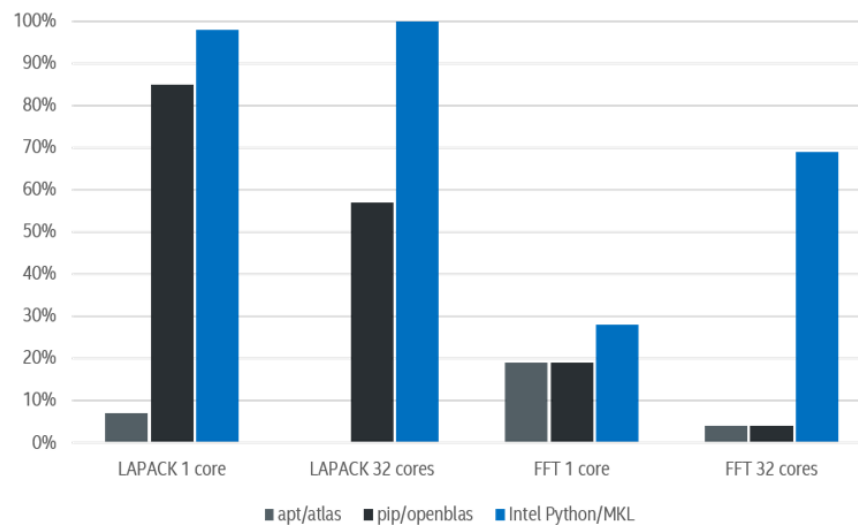
На первой неделе сентября этого года вышла в свет новая версия продукта Intel Parallel Studio XE 2019.

Ещё с версии XE 2016 таковых имелось целых 4 штуки:

- быстрая математическая библиотека Intel Math Kernel Library (MKL)
- библиотека для обработки данных и мультимедиа Intel Integrated Performance Primitives (IPP)
- библиотека шаблонов C++ Intel Threading Building Blocks (TBB)
- библиотека для машинного обучения и аналитики данных Intel Data Analytics Acceleration Library (DAAL).

Важное дополнение во всех версиях студии (и не только) – появился пакет Intel Distribution for Python\*, который позволяет использовать Python (2.7 и 3.5), «оптимизированный» заботливыми инженерами компании Intel, на Windows, Linux и OS X. В самом языке ничего нового не появилось, просто пакеты NumPy, SciPy, pandas, scikit-learn, Jupyter, matplotlib, и mpi4py теперь оптимизированы под «железо» с использованием библиотек Intel MKL, TBB, DAAL и MPI, за счёт чего приложение выполняется более эффективно и быстро.

Производительность Python в процентах относительно C/Intel MKL для процессоров Xeon



Конфигурация: apt/atlas: installed with apt-get, Ubuntu 16.10, python 3.5.2, numpy 1.11.0, scipy 0.17.0; pip/openblas: installed with pip, Ubuntu 16.10, python 3.5.2, numpy 1.11.1, scipy 0.18.0; Intel Python: Intel Distribution for Python 2017; Hardware: Xeon: Intel Xeon CPU E5-2698 v3 @ 2.30 GHz (2 sockets, 16 cores each, HT=off), 64 GB of RAM, 8 DIMMS of 8GB@2133MHz; Xeon Phi: Intel Intel® Xeon Phi™ CPU 7210 1.30 GHz, 96 GB of RAM, 6 DIMMS of 16GB@1200MHz

Появилось:

- Библиотека шаблонов C++ SIMD Data Layout Templates (SDLT) для решения проблемы перехода от массива структур (AoS – Array of Structures) к структуре массивов (SoA – Structure of Arrays), что приводит к уменьшению непоследовательных доступов к памяти и gather/scatter инструкций при векторизации.
- Поддержка процессоров второго поколения Intel Xeon Phi и новое имя `mic_avx512`.
- Ещё более полная поддержка OpenMP (как стандарта 4.0, так и более позднего 4.5).
- Новый функционал по оффлоду на Intel Xeon Phi с помощью OpenMP
- Возможность генерировать файлы (html или текстовые) с исходным кодом и интегрированным отчетом об оптимизации, что будет очень полезно разработчикам, работающим из командной строки без среды разработки. Кроме того, информативность самих отчетов об оптимизации и векторизации улучшена по многим направлениям.
- Новый атрибут, директива и опция компилятора для выравнивания кода (не данных, а именно самих инструкций для функций или циклов).

- Более широкая поддержка стандарта C++14, в частности теперь мы можем использовать шаблоны переменных, освобождать память определенного размера глобальным оператором `delete`, пользоваться `constexpr` функциями с заметно меньшими ограничениями, чем было в стандарте C++11. Полный список поддерживаемых возможностей стандарта можно найти на этой страничке.
- Стандарт C11 поддерживается теперь полностью (не путать с C++11, который уже давно поддерживается), за исключением ключевого слова `_Atomic` и соответствующего атрибута `__attribute__((atomic))`.
- Появился ряд новых опций компилятора, призванных упростить его использование. Например, ключик `/fp:consistent` (Windows) и `-fp-model consistent` (Linux), который включает в себя ряд других (`/fp:precise` `/Qimf-arch-consistency:true` `/Qfma-`).
- Новые возможности по оффлоду вычислений на интегрированную графику с использованием OpenMP. Например, теперь возможен асинхронный оффлод с помощью клаузы `DEPEND` в `TARGET` директиве. Кроме этого, компилятор может производить векторизацию с типом `short`.

В целом, компилятор, как и ожидалось, стал генерировать ещё более производительный код, поддерживая все последние новшества в «железе».

### 3.5 OpenMP

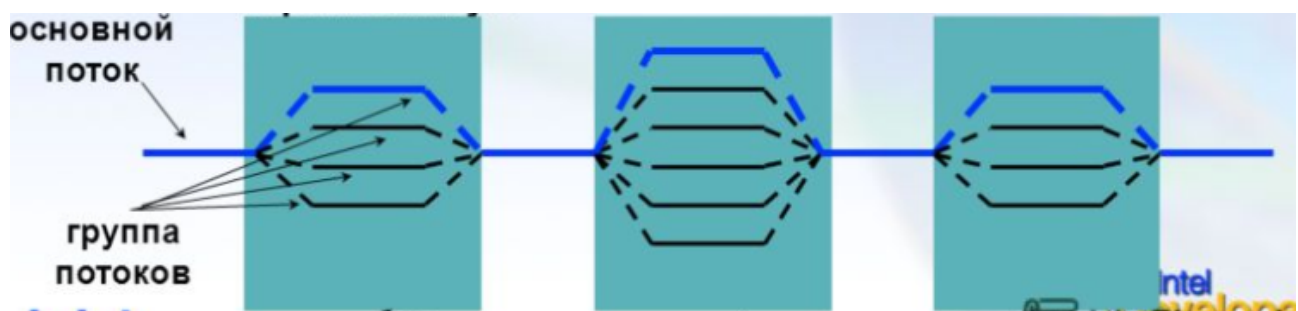
OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций. Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran. Поддерживается производителями аппаратуры (Intel, HP, SGI, Sun, IBM), разработчиками компиляторов (Intel, Microsoft, KAI, PGI, PSR, APR, Absoft).

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

Задачи, выполняемые потоками параллельно, также как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм. Например, участок кода на языке Fortran, который должен исполняться несколькими потоками, каждый из которых имеет свою копию переменной N, предваряется следующей директивой: `!$OMP PARALLEL PRIVATE(N)`

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

Основной поток порождает дочерние потоки по мере необходимости. Модель `fork-join`.



Синтаксис. В основном конструкции OpenMP – это директивы компилятора

Параллельные регионы. Там, где задан этот регион программа исполняется параллельно.

Модель исполнения. Существует две модели исполнения: динамическая, когда количество используемых потоков в программе может варьироваться от одной области параллельного выполнения к другой, и статическая, когда количество потоков фиксировано. Модель исполнения контролируется или через переменную окружения `OMP_DYNAMIC` или с помощью вызова функции `omp_set_dynamic()`.

Ключевыми элементами OpenMP являются:

- конструкции для создания потоков (директива `parallel`),
- конструкции распределения работы между потоками (директивы `DO/for` и `section`),
- конструкции для управления работой с данными (выражения `shared` и `private` для определения класса памяти переменных),
- конструкции для синхронизации потоков (директивы `critical`, `atomic` и `barrier`),

- процедуры библиотеки поддержки времени выполнения (например, `omp_get_thread_num`),
- переменные окружения (например, `OMP_NUM_THREADS`).

Библиотечные функции OpenMP. Для эффективного использования процессорного времени компьютера и написания гибких программ пользователю предоставляется возможность управлять ходом выполнения программы посредством библиотечных функций.

Преимущества:

- Легкий путь использования Hyper-Threading
- Переносимость и стандартизация
- Специальные инструменты профилирования

Таким образом, можно увеличить количество выполняемых потоков для процессоров, поддерживающим технологию Hyper-Threading.

- Необходимо узнать поддерживает ли ваш процессор данную технологию;
- Включить в настройках BIOS данную технологию;
- В зависимости от ОС получить информацию о количестве логических процессоров после подключения данной технологии;
- Подсчитать количество потоков оптимальных для выполнения по полученной информации;

При работе с многопоточными приложениями возникают большие трудности при отладке таких приложений такие как, сбои и утечки памяти, вызванные параллельным доступом, взаимоблокировки и состояния гонки между потоками. Одним из самых популярных IDE для разработки многопоточных приложений является Intel Parallel Studio с использованием C++.

### 3.6. Разработки Intel

**Intel Parallel Studio** — пакет разработки распараллеленного программного обеспечения разработанный фирмой Intel, сочетающий в себе ведущие в индустрии C/C++ компилятор и Fortran компилятор, различные библиотеки, инструменты профилирования и многое другое. В комплект поставки входят программные продукты:

- Intel Parallel Advisor,

- Intel Parallel Composer,
- Intel Parallel Inspector,
- Intel VTune Amplifier.

## **Intel Parallel Advisor**

Существуют два похода к написанию параллельных програм. Первый - это распараллеливание, частичное или полностью, уже готовых последовательных приложений для ускорения работы некоторых достаточно изолированных участков, часто алгоритмов, которые не затрагивают всю архитектуру проекта. Здесь методологии как таковой не существует. Разработчик просто анализирует приложение и определяет участки программы, которые потребляют максимальное количество ресурсов микропроцессора. Затем анализируется структура проекта и принимается решение о модификации того или иного алгоритма. Второй подход предполагает изначальный дизайн с учетом требований параллельного выполнения нагрузки. И если концептуально проект можно разделить на участки, которые должны выполняться одновременно, то начать его реализацию в виде программы часто является сложной задачей для новичков. Особенно трудно написать проект так, чтобы потом не пришлось прибегать к первому подходу.

Здесь на помощь придет Parallel Advisor. Это совершенно новая категория или новый класс инструментов, который несет в себе некую методологию создания параллельных программ «с нуля» с использованием правильных подходов к их реализации, в том числе с использованием параллельных библиотек производительности. Понятно, что не каждая программа или алгоритм легко и просто параллелятся. Advisor найдет, из-за чего именно параллельная реализация может оказаться неэффективной, и попытается выдать нужные решения. Кроме того, все знания по применению параллельных библиотек будут собраны здесь в виде сэмплов и шаблонов, для того чтобы максимально облегчить начальный этап их использования. Однако и в случае с

распараллеливанием готовой последовательной программы Advisor предоставит «путеводную нить», или workflow, по распараллеливанию, проверке корректности и оптимизации приложения, если разработчик пока еще не в силах «уложить» в голове эту методологию. Но через какое-то время работы с проектом надобность в Advisor'е отпадает, так как становятся понятными принципы разработки параллельной программы, а элементы методологии рассматриваются как дополнительный Help Page ресурс.

В текущем пакете Advisor не присутствует, зато его первая реализация в виде утилиты Intel Parallel Advisor Ligt доступна среди других исследовательских проектов на сайте Intel. Это сделано для того, чтобы сообщество разработчиков могло поделиться своими идеями относительно методологий распараллеливания, которые могут быть потом реализованы в будущем продукте. Такой же путь уже прошли хорошо известные теперь параллельная библиотека TBV и инструмент Performance Tuning Utility, которые стали частью Intel Parallel Studio, возвращая разработчикам их идеи в виде open source библиотеки для разработки на C++ и совершенно нового продукта для оптимизации производительности.

### **Intel Parallel Composer**

Несмотря на некий маркетинговый акцент в названии продукта Composer – это не просто компилятор C++ от Intel. Он уже проинтегрирован в Visual Studio вместе с библиотекой производительности IPP и параллельной библиотекой TBV, что значительно облегчает процесс разработки параллельного кода для новичков, т.е. тех, кто еще не пользовался продуктами Intel, такими, например, как Compiler Pro, и только собирается попробовать улучшить производительность своих приложений с помощью технологий Intel.

Наличие сразу нескольких компонент в пакете позволит сразу же начать оптимизировать свою программу с использованием параллельных технологий, которые содержит Composer:



- Вычислительные примитивы, реализованные в виде функций в библиотеке IPP, гарантируют высокую производительность алгоритмов на платформах Intel;
- Поддержка новой версии стандарта OpenMP 3.0 позволит использовать multitasking, недоступный в предыдущих версиях, которые поддерживаются в том числе и компилятором Microsoft;
- Новый тип данных Valarray немного упростит код, реализующий векторные операции, а компилятор сгенерирует эффективный бинарный код, задействующий SIMD-инструкции для увеличения производительности;
- Поддержка компилятором элементов стандарта C++ 0x облегчит кодирование программистам.

## **Intel Parallel Inspector**

Это, пожалуй, самый востребованный и ожидаемый инструмент на сегодняшний день, так как он помогает избавиться от ошибок в многопоточной программе на этапе верификации, повышая корректность и стабильность ее исполнения. Несмотря на свой характерный функционал Inspector применяется не только командами тестировщиков (QA team). Нормальная инженерная практика предполагает проверку программы на наличие ошибок и самим разработчиком, хотя бы на уровне юнит-тестов (unit tests).

Давайте разберем, какие же ошибки помогает обнаружить Parallel Inspector. Инструмент адресует два класса ошибок: ошибки многопоточности и ошибки работы с памятью, причем анализ для каждого класса запускается отдельно. Последний класс ошибок хорошо известен программистам, которые до последнего времени использовали различные инструменты, чтобы найти утечки памяти, нарушение целостности стека или доступ по несуществующим адресам. Второй класс ошибок связан с многопоточной природой програм. Они неизбежно возникают при разработке параллельных приложений, и их

чрезвычайно сложно отловить, особенно если они проявляются нерегулярно и только при совпадении определенных условий.

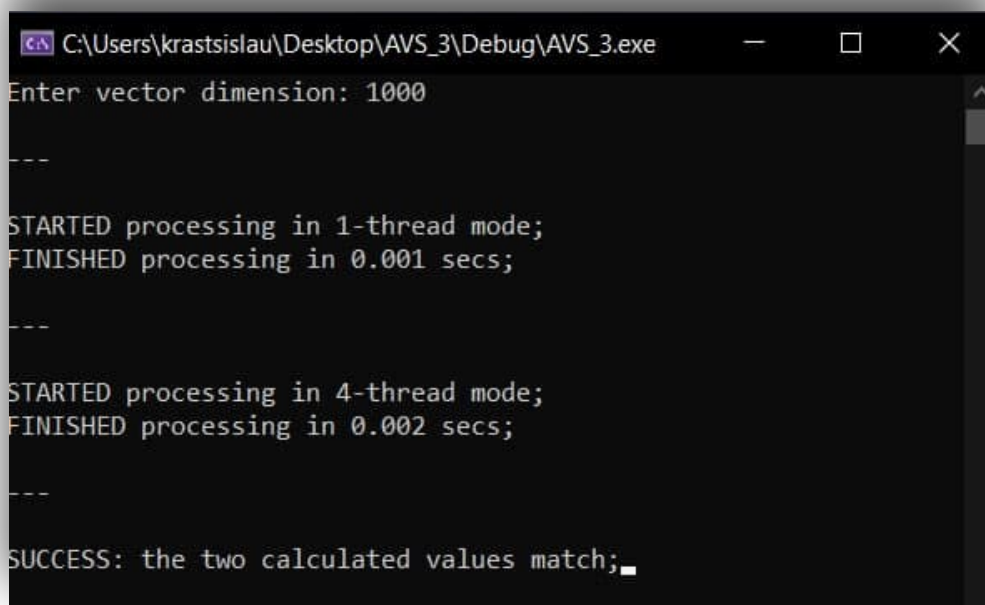
Под мультизадачностью подразумевают способность компьютера выполнять несколько задач одновременно. На самом деле процессор некоторое время выполняет один командный поток, затем быстро переключается на второй и выполняет его, переключается на третий и т.д. При этом при каждом переключении сохраняется контекст прерываемого потока, так что потом процессор сможет "безболезненно" продолжить выполнение прерванного потока команд. Благодаря высокому быстродействию создается иллюзия того, что все задачи выполняются одновременно (параллельно). Для управления мультизадачностью нет специальных команд. Задачи переключаются командами FAR CALL, FAR JMP, INT, IRET. Однако при этом участвуют специальные дескрипторы: дескриптор сегмента состояния задачи (Task State Segment) и дескриптор шлюза задачи. Когда управление передается на один из таких дескрипторов, происходит переключение задачи. При переключении задачи процессор сохраняет (восстанавливает) свой контекст в сегменте состояния задачи (TSS). Селектор TSS выполняемой задачи хранится в регистре задачи (Task Register). При переключении задачи процессор может сменить LDT, что позволяет назначить каждой задаче свое адресное пространство, недоступное для других задач. Можно также перегрузить CR3, что позволяет применить для изолирования задач механизм страничного преобразования. Дескриптор TSS относится к системным дескрипторам. Поле Type дескриптора TSS может содержать код 1001, если это доступный TSS, или 1011, если это занятый TSS, т.е. если задача активна в настоящий момент. На рис. 11 представлен формат сегмента TSS для процессора i80386. Из рисунка видно, что в TSS предусмотрены поля для хранения сегментных регистров GS, FS, DS, SS, CS, ES. Имеется поле для хранения содержимого регистра LDTR, указывающего на локальную таблицу дескрипторов, распределённую для данной задачи. Для хранения содержимого 32-разрядных регистров используются поля 25 TSS, обозначенные на рис.11 как EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX,

EFLAGS, EIP. Поле CR3 хранит содержимое системного регистра CR3. Этот регистр является указателем на каталог таблиц страниц. Таким образом, каждая задача может иметь свой собственный каталог таблиц страниц, что позволяет выполнить изоляцию задач не только на уровне сегментов, но и на уровне страниц. Сегмент TSS процессора i80386 TSS процессора i80386 содержит указатели на стеки для второго, первого и нулевого приоритетных колец. Это поля SS2:ESP2, SS1:ESP1, SS0:ESP0. Поле LINK используется для ссылки на TSS, вызвавшей задачи при вложенном вызове задач, аналогично тому, как это было в процессоре i80286. 26 Бит T используется для отладки. Если он установлен в "1", при переключении на задачу возникает отладочное исключение, которое может быть использовано системным отладчиком. Для обеспечения безопасной работы системы необходимо ограничить доступ программам пользователя ко всем или по крайней мере к некоторым портам ввода/вывода. Злонамеренная программа, имеющая доступ к портам контроллера прямого доступа к памяти, может выполнить с помощью этого контроллера чтение или запись информации по любым физическим адресам. Процессор i80286 хранит в регистре флагов уровень привилегий IOPL, на котором разрешено выполнять команды ввода/вывода. С помощью этого механизма можно запретить непривилегированным программам выполнять команды ввода/вывода. Однако такой способ защиты не слишком удобен. Некоторые порты ввода/вывода не только безопасны для использования, но и весьма полезны для обычных программ (например, порт системного динамика или принтера). Битовая карта ввода/вывода процессора i80386 позволяет для каждой задачи определить порты, которые эта задача может использовать. То есть операционная система имеет возможность санкционировать любую задачу для использования любого набора адресов портов ввода/вывода. Если задача попытается обратиться к несанкционированному порту ввода/вывода, произойдет исключение. Сегмент TSS содержит поле, обозначенное на рис.11. как база карты ввода/вывода. Оно ввода/вывода задачи, использующей данный TSS. Поле базы карты ввода/вывода указывает 16-разрядное смещение относительно TSS. Предел TSS должен определяться с учетом карты. Каждый

бит в карте ввода/вывода соответствует адресу байта порта ввода/вывода (карта состоит из 64 Кбит для описания доступа к 65536 портам). После битовой карты должен располагаться байт OFFh. При выполнении 16- или 32-разрядных операций ввода/вывода процессор проверяет все биты (2 или 4 бита), соответствующие адресу порта. Если проверяемый бит установлен в “1”, происходит исключение. Для привилегированных программ, если уровень привилегий меньше или равен уровню IOPL, процессор не выполняет проверку битовой карты ввода/вывода. Чтобы полностью запретить задаче обращаться к портам ввода/вывода, достаточно установить базу карты ввода/вывода большей или равной пределу TSS. В этом случае любая команда ввода/вывода приведёт к генерации исключения. Лишь значение определены. Именно это число является минимальным размером TSS. Операционная система может по своему усмотрению устанавливать размер TSS и включать в сегмент TSS дополнительную информацию, необходимую для работы задачи и зависящую от конкретной операционной системы (например, контекст сопроцессора, указатели открытых файлов или указатели на именованные службы для указания расположения битовой карты начала битовой карты ввода/вывода первых 68h байт сегмента состояния задачи строго 27 конвейеры сетевого обмена). Включенная в автоматически заменяется процессором при выполнении команды CALL или JMP, селектор которой указывает на дескриптор сегмента TSS в таблице GDT (дескрипторы этого типа могут быть расположены только в этой таблице). При переключении задачи с помощью прерывания или особого случая происходит автоматический возврат к прерванной задаче. Однако организуя вложение задач, необходимо помнить, что, в переключении задачи в стек ничего не включается. Дескриптор TSS задачи, выполняемой в данный момент, помечается как занятый. При переключении на другую задачу с вложением (по INT или FAR CALL) дескриптор TSS остается помеченным. Переключиться на занятую задачу нельзя (возникает нарушение общей защиты - исключение №13). Для переключения задач также действуют правила привилегий. По команде JMP или CALL можно переключиться на задачу, TSS которой менее привилегирован:

этот сегмент информация отличие от процедур, при  $DPLTSS \geq \max(CPL, RPL)$ . Для особых случаев и прерываний это правило не действует. Если обработчик прерывания выполнен в виде отдельной задачи, то он может быть вызван независимо от значения CPL. Не совсем удобно адресовать именно TSS для переключения задачи, т.к., во- первых, TSS могут быть размещены только в GDT (а в IDT или LDT - нет), а, во- вторых, если пользоваться только TSS, то каждую задачу мы "намертво" привязываем к определенному уровню привилегий (DPLTSS), с которого она доступна для переключения. Этих недостатков лишены шлюзы задачи. Формат дескриптора шлюза задачи приведен. Дескриптор шлюза задачи для процессора i80386 Шлюз задачи содержит селектор TSS. Шлюзы задач можно размещать и в IDT, что позволяет выполнять обработчики прерываний в виде отдельных задач, и 28 в LDT, что позволяет более гибко управлять переключением задач: для второй задачи первая может быть видна с одного уровня привилегий, а для третьей - с другого. Последняя возможность обеспечивается особым правилом привилегий: при переключении задачи через шлюз учитывается только DPLшлюза, а DPLTSS не играет роли, поэтому одной задаче может соответствовать множество шлюзов с различными DPL.

#### 4. Демонстрация работы программы



```
C:\Users\krastislau\Desktop\AVS_3\Debug\AVS_3.exe
Enter vector dimension: 1000

---

STARTED processing in 1-thread mode;
FINISHED processing in 0.001 secs;

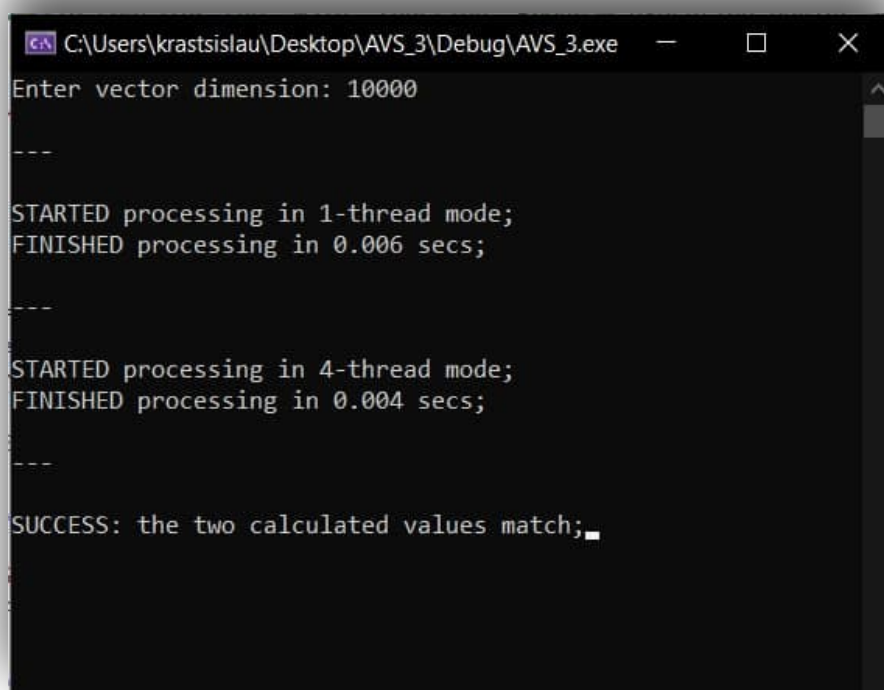
---

STARTED processing in 4-thread mode;
FINISHED processing in 0.002 secs;

---

SUCCESS: the two calculated values match;
```

Рисунок 1. Результат выполнения программы для вектора с количеством координат равным  $10^3$



```
C:\Users\krastislau\Desktop\AVS_3\Debug\AVS_3.exe
Enter vector dimension: 10000

---

STARTED processing in 1-thread mode;
FINISHED processing in 0.006 secs;

---

STARTED processing in 4-thread mode;
FINISHED processing in 0.004 secs;

---

SUCCESS: the two calculated values match;
```

Рисунок 2. Результат выполнения программы для вектора с количеством координат равным  $10^4$

```
C:\Users\krastislau\Desktop\AVS_3\Debug\AVS_3.exe
Enter vector dimension: 100000

---

STARTED processing in 1-thread mode;
FINISHED processing in 0.059 secs;

---

STARTED processing in 4-thread mode;
FINISHED processing in 0.019 secs;

---

SUCCESS: the two calculated values match;
```

Рисунок 3. Результат выполнения программы для вектора с количеством координат равным  $10^5$

```
C:\Users\krastislau\Desktop\AVS_3\Debug\AVS_3.exe
Enter vector dimension: 1000000

---

STARTED processing in 1-thread mode;
FINISHED processing in 0.438 secs;

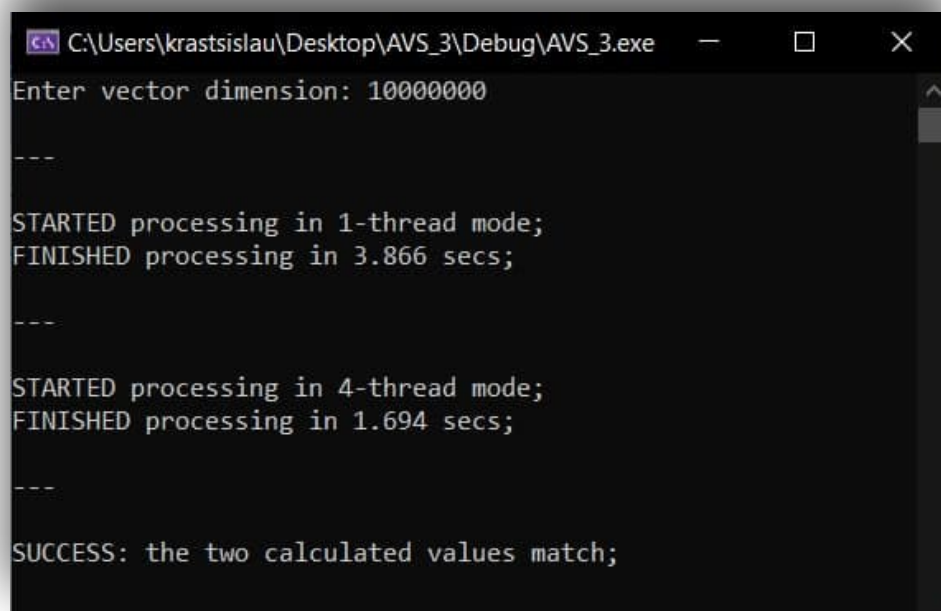
---

STARTED processing in 4-thread mode;
FINISHED processing in 0.182 secs;

---

SUCCESS: the two calculated values match;
```

Рисунок 4. Результат выполнения программы для вектора с количеством координат равным  $10^6$



```
C:\Users\krastisla\l\Desktop\AVS_3\Debug\AVS_3.exe
Enter vector dimension: 10000000

---

STARTED processing in 1-thread mode;
FINISHED processing in 3.866 secs;

---

STARTED processing in 4-thread mode;
FINISHED processing in 1.694 secs;

---

SUCCESS: the two calculated values match;
```

Рисунок 5. Результат выполнения программы для вектора с количеством координат равным  $10^7$



## 5. Графики

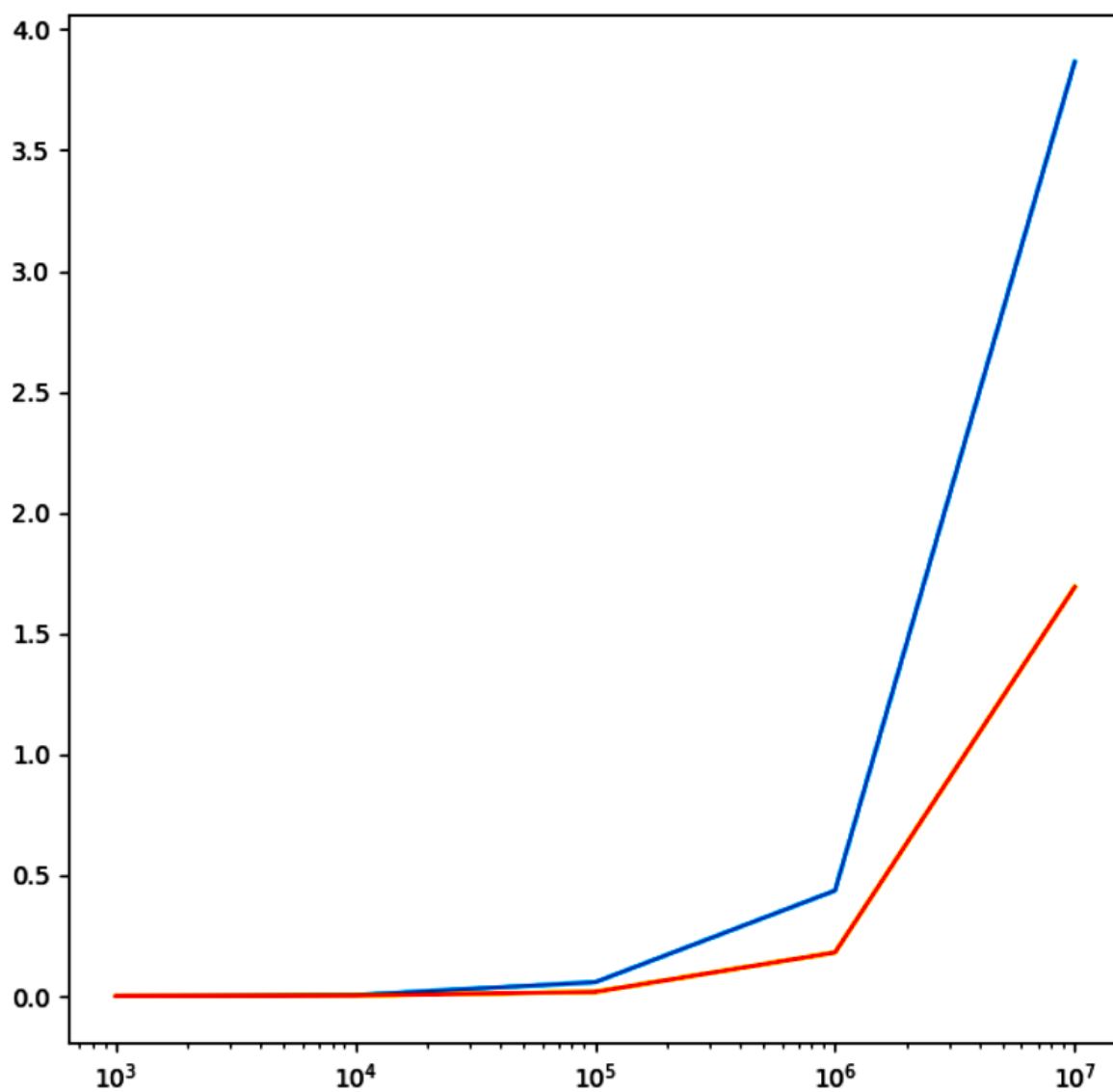


График 1. Зависимость времени выполнения в секундах от числа координат вектора (оранжевый график – 1 поток, синий график – 4 потока)

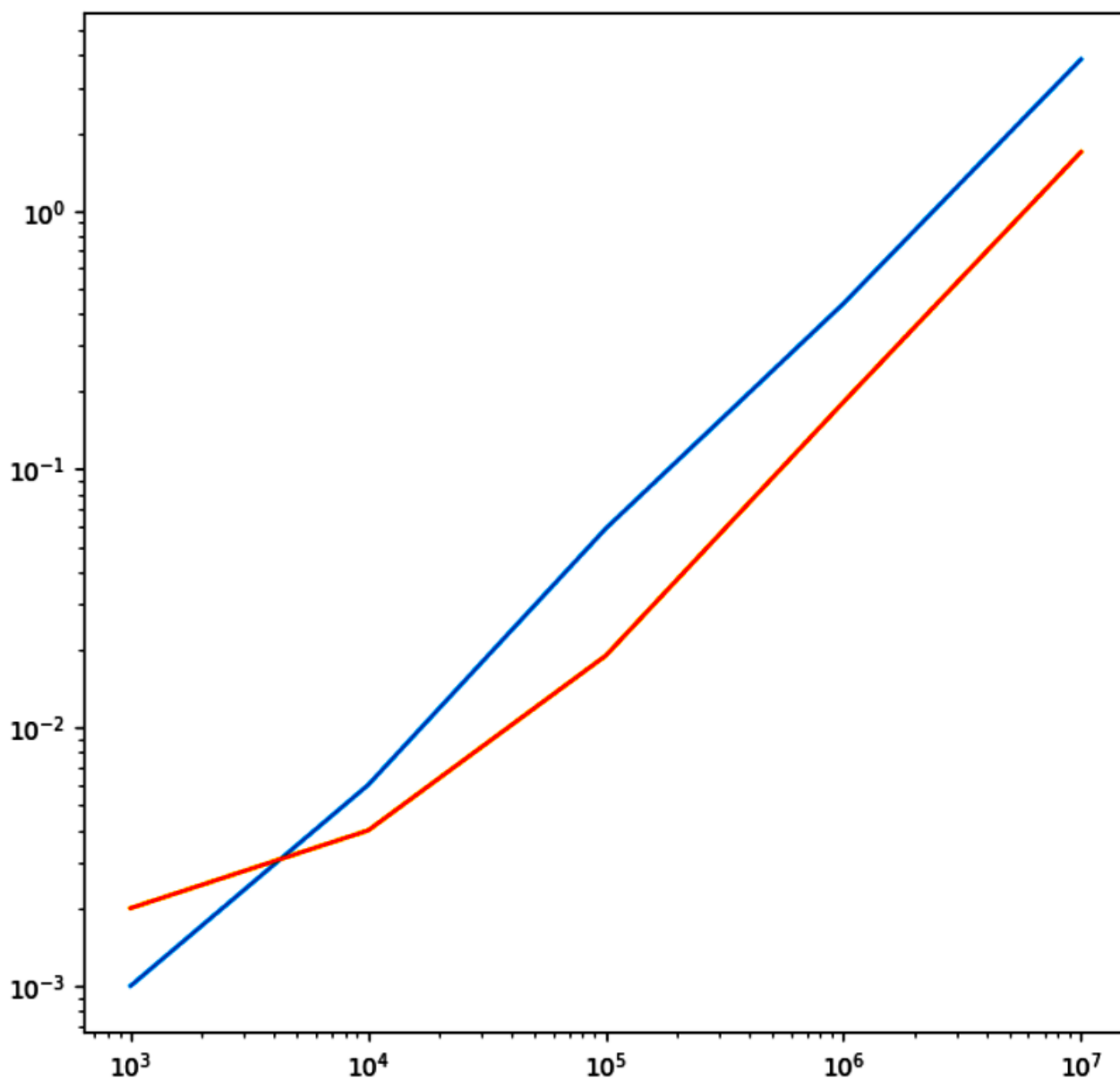


График 2. Зависимость времени выполнения в секундах от числа координат вектора (оранжевый график – 1 поток, синий график – 4 потока), обе шкалы логарифмические

## **Вывод**

В результате выполнения лабораторной работы изучено API, предназначенное для программирования многопоточных приложений на многопроцессорных системах с общей памятью. Для закрепления теоретических знаний была разработана программа, вычисляющая скалярное умножение двух многомерных векторов. Для ее реализации был использован интерфейс OpenMP для параллельного выполнения частей программы. Для сравнения так же был разработан однопоточный вариант алгоритма. Сравнение показало, что программа, работающая в 4-х потоках, производит вычисления быстрее однопоточной для векторов, количество координат которых больше  $10^4$ . Для векторов, количество координат которых меньше  $10^4$  работает быстрее однопоточный алгоритм. Таким образом, можно сделать вывод, что использование многопоточности на больших объемах данных позволяет ускорить вычисления. Однако, если количество данных небольшое, то многопоточный алгоритм проигрывает однопоточному из-за затрат времени на переключение контекстов.

## **Приложение 1. Исходный код программы**

Исходный код программы доступен по ссылке: [Stanislav-3 \(Stanislav Korenevsky - github.com\)](#);