МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение
высшего профессионального образования -
«Оренбургский государственный университет»

Кафедра программного обеспечения вычислительной
техники и автоматизированных систем

А.Ю. ВЛАДОВА

РАЗРАБОТКА МАСШТАБИРУЕМЫХ ПРОГРАММ ДЛЯ МНОГОЯДЕРНЫХ АРХИТЕКТУР

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано к изданию Редакционно – издательским советом государственного образовательного учреждения высшего профессионального образования - «Оренбургский государственный университет»

Оренбург 2006

ББК 32.973-02я73
В 57
УДК 004.72(076.5)

Рецензенты
доктор технических наук, профессор Н. А. Соловьев,
кандидат технических наук, доцент Ю. А. Кудинов

Владова А.Ю.

В 57 Разработка масштабируемых программ для многоядерных архитектур: лабораторный практикум/ А. Ю. Владова - Оренбург: ГОУ ОГУ, 2006. – 28 с.: ил.

В лабораторном практикуме изложены предпосылки разработки и основы многоядерной архитектуры на примере процессоров Intel и AMD, особенности проектирования и написания многопоточных программ с акцентом на причины их плохой масштабируемости. Практикум содержит методические указания к выполнению лабораторных и самостоятельных работ.

Лабораторный практикум подготовлен на кафедре «Программное обеспечение вычислительных средств и автоматизированных систем» и предназначен для студентов старших курсов специальности ПОВТАС по дисциплинам «Архитектуры вычислительных систем и сетей» и «Программное обеспечение сетей ЭВМ».

ББК 32.973-02я73

Содержание

Введение

В лабораторном практикуме рассматривается круг вопросов, связанных с основами разработки многопоточных приложений, элементами их инструментированной отладки и тестирования. Целью лабораторного практикума является развитие у студентов-программистов навыков разработки масштабируемых многопоточных программ. Для достижения поставленной цели рассмотрены следующие задачи:

- основы многоядерной архитектуры;
- проектирование многопоточных программ с использованием различных инструментальных средств;
- причины плохой масштабируемости программ;
- теоретическая и практическая оценка ускорения и масштабируемости программ;
- элементы оптимизации.

Для демонстрации влияния пропускной способности шины на плохую масштабируемость программ разработана лабораторная работа №1, в которой описаны реализация последовательного и многопоточного приложений, демонстрирующих проблему, предварительная и практическая оценки ускорения и масштабируемости, оптимизация программы для повышения доли параллельного кода, а также инструментированный анализ загрузки шины.

Влияние размера пула потоков на масштабируемость программ раскрывается в лабораторной работе №2, в которой приводятся причины использования пула потоков, математическое описание системы, анализ предполагаемых проблем при оптимизации приложения, теоретическая и практическая оценки ускорения и масштабируемости, даны рекомендации по выбору размера пула и очереди.

В качестве языков программирования, в которых разрабатывались и отлаживались приведенные примеры, выбраны C++ и C#, как мощные современные инструменты создания многопоточных приложений. Все теоретические сведения подкреплены примерами и графиками, которые могут служить базой при разработке полноценного программного обеспечения в данной области.

Отчет по каждой лабораторной работе должен включать титульный лист, постановку задачи, теоретические сведения, иерархическую схему процедур, текст основных процедур, результаты работы и выводы.

1 Предпосылки создания многоядерной архитектуры

Гонка тактовых частот, продолжавшаяся на протяжении многих лет уходит в прошлое, так как развитие физики полупроводников, следствием которого стало увеличение числа логических элементов на единицу площади, подчиняющееся закону Гордона Мура, провоцирует ряд проблем, таких как перегрев и физическое ограничение плотности транзисторов. Проблема перегрева транзисторов в современных процессорах стоит очень остро. Сами транзисторы становятся все меньше, но при этом с ростом тактовой частоты процессора они потребляют больше мощности и выделяют больше тепла. До бесконечности это продолжаться не может, поскольку приводит к быстрому обгоранию соответствующего контакта разъема, нагреву и сколам процессора, разряду батарей питания. Суть проблемы заключается в том, что современные процессоры потребляют такую мощность, что при питании их стабилизаторов от шины +5В ток превосходит разумные пределы (мощность равна произведению тока на напряжение, поэтому, чем ниже напряжение, тем выше ток при той же мощности). По всей видимости, масштабирование процессоров по тактовой частоте оказалось не столь простой задачей, как предполагалось, и потому при сегодняшних технологических нормах производства процессоров и малоэффективных воздушных системах охлаждения добиться линейного масштабирования тактовой частоты процессоров не удаётся.

Таким образом, производители столкнулись с проблемой достижения предела прогнозируемой скорости роста тактовой частоты – небольшого зазора оптимизации исполнения в пределах одного ядра, за которым рост сложности становится неоправдан и для дальнейшего наращивания размеров кэш-памяти процессоров остается лишь небольшой зазор для развития [1]. Осознав, что увеличение прежними темпами тактовых частот процессоров не представляется невозможным, нужно было искать принципиально иные технологии увеличения производительности процессоров. Достигая пределов роста скорости и уровня охлаждения, производители процессоров пришли к одному и тому же решению: объединить в одной микросхеме два процессорных ядра. Поэтому перспективным направлением для дальнейшего развития видится многоядерная и многопотоковая архитектура процессоров.

2 Преимущества многоядерной архитектуры процессоров

Еще несколько лет назад в силу технологических ограничений все многопотоковые процессоры строились на базе одного ядра, и такая многопотоковость была названа «одновременной» — Simultaneous Multi Threading (SMT), но с появлением многоядерных процессоров появился альтернативный тип многопотоковости — Chip Multi Processor (CMP).

Многоядерная архитектура — это архитектура, в составе которой находится один физический процессор, содержащий основные логические узлы нескольких процессоров.

Преимущество такого процессора над одноядерным проявляется, прежде всего, при работе с многопоточными приложениями. Многопоточные задачи работают быстрее на двухядерных процессорах, потому что операционная система может распределять программные потоки отдельно по каждому ядру, в то время как на одноядерных процессорах задачи меняются по мере выполнения, то есть по очереди. Применение этой технологии позволит увеличить производительность процессоров нового поколения и одновременно избежать роста потребления энергии, которое накладывает ограничения на развитие одноядерных процессоров. Поскольку производительность памяти увеличивается медленнее, чем скорость процессоров, то чем выше частота процессора, тем больше потеря производительности при обращении к памяти. Два ядра получаются предпочтительней, чем одно, так как в этом случае легче обеспечить процессор данными для обработки.

Такая архитектура предоставляет одну из возможностей преодоления врожденной слабости современных процессоров — «бутылочного горла» архитектуры фон Неймана в соответствии с рисунком 1.

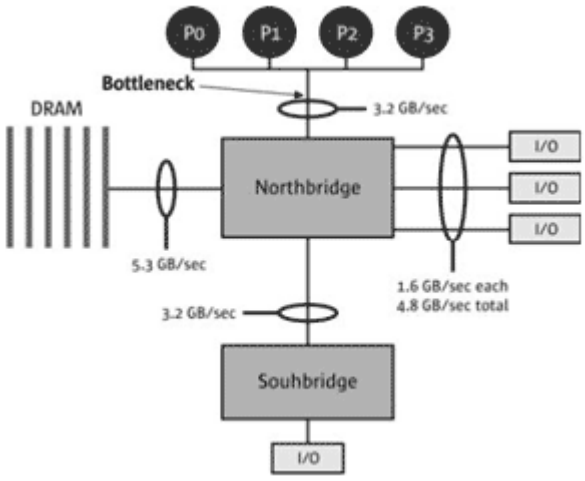


Рисунок 1 – Демонстрация «узкого горла» архитектуры фон Неймана на примере четырехпроцессорной архитектуры Intel Xeon

Впервые публично об этом недостатке сказал создатель Fortran Джон Бэкус на церемонии вручения ему премии имени Тьюринга в 1977 году: «Что такое

компьютер по фон Нейману? Когда 30 лет назад Джон фон Нейман и другие предложили свою оригинальную архитектуру, идея показалась элегантной, практичной и позволяющей упростить решение целого ряда инженерных и программистских задач. И хотя за прошедшее время условия, существовавшие на момент ее публикации радикально изменились, мы отождествляем наши представления о компьютерах с этой старой концепций. В простейшем изложении фон-неймановский компьютер состоит из трех частей: это центральный процессор, память и соединяющий их канал, который служит для обмена данными между CPU и памятью, причем маленькими порциями - по одному слову.

Я предлагаю назвать этот канал «бутылочным горлом фон Неймана». Наверняка должно быть менее примитивное решение, чем перекачивание огромного количества данных через «узкое бутылочное горло». Такой канал не только создает проблему для трафика, но еще и является «интеллектуальным бутылочным горлом», которое навязывает программистам «пословное» мышление, не позволяя рассуждать в более высоких концептуальных категориях» [1].

С момента произнесения Бэкусом его речи в программировании произошли заметные сдвиги, появились функциональные и объектно-ориентированные технологии, и с их помощью удалось преодолеть то, что Бэкус называл «интеллектуальным фон-неймановским бутылочным горлом». Однако архитектурная первопричина данного явления, врожденная болезнь канала между памятью и процессором — его ограниченная пропускная способность — не исчезла, несмотря на прогресс в области технологии за прошедшие с тех пор 30 лет. С годами эта проблема постоянно усугубляется, поскольку скорость работы памяти растет гораздо медленнее, чем производительность процессоров, и разрыв между ними становится все больше .

Фон-неймановская архитектура компьютера не является единственно возможной. С точки зрения организации обмена командами между процессором и памятью все компьютеры можно разделить на четыре класса (классификация Флинна) [11]:

SISD (Single Instruction Single Data) — «один поток команд, один поток данных»»;

SIMD (Single Instruction Multiply Data) — один поток команд, много потоков данных;

MISD (Multiple Instruction Single Data) — много потоков команд, один поток данных;

MIMD (Multiple Instruction Multiple Data) — много потоков команд, много потоков данных.

Из этой классификации видно, что фон-неймановская машина является частным случаем, попадающим в категорию SISD. Возможные усовершенствования в рамках архитектуры SISD ограничиваются включением в нее конвейеров и других дополнительных функциональных узлов, а также использованием разных методов кэширования. Две другие категории архитектур (SIMD, в которую входят векторные процессоры, и конвейерные архитектуры MISD) были реализованы в нескольких проектах, но не стали массовыми. Если оставаться в рамках

этой классификации, то единственной возможностью преодоления ограничений «бутылочного горла» остается развитие архитектур класса MIMD. В их рамках обнаруживается множество подходов: это могут быть и различные параллельные и кластерные архитектуры, и многопотоковые процессоры.

Широкое распространение мультиядерных технологий - это следующий этап развития компьютерных технологий, который преобразит существующую вычислительную среду. Основные выгоды, получаемые от внедрения мультиядерных технологий представляются следующими.

Для коммерческого рынка - корпоративные IT-системы получат значительное увеличение производительности, используя оптимизированные мультипоточные приложения. Внедрение новых процессоров позволит создавать более сложные системы, с минимальными вложениями, опираясь на существующую инфраструктуру. Кроме того, такие системы отличаются простым управлением, упрощенным менеджментом, низкой совокупной стоимостью владения, высокой эффективностью и производительностью.

Для бизнеса и конечных пользователей - мультиядерная технология AMD способна существенно увеличить количество выполняемой работы в задачах требующих интенсивных вычислений, например таких, как мультимедиа, создание цифрового контента, обеспечения безопасности и других.

Разработчики и тестеры программного обеспечения озадачены созданием новых алгоритмов, обрабатывающих данные одновременно. [2].

Конечно, говорить о том, что двухядерные процессоры в два раза производительнее одноядерных, не приходится. Причина заключается в том, что для реализации параллельного выполнения двух потоков необходимо, чтобы эти потоки были полностью или частично независимы друг от друга, а кроме того, чтобы операционная система и само приложение поддерживали на программном уровне возможность распараллеливания задач. И в связи с этим стоит подчеркнуть, что сегодня далеко не все приложения удовлетворяют этим требованиям и потому не смогут получить выигрыша от использования двухядерных процессоров. Впрочем, уже сегодня существует немало приложений, которые оптимизированы для выполнения в многопроцессорной среде, и такие приложения, несомненно, позволят использовать преимущества двухядерного процессора. Кроме того, она позволяет выявить преимущества при одновременной работе с несколькими приложениями, что является типичной ситуацией на сегодняшний день.

2.1 Архитектура Intel Core

Рассмотрим технологию Intel Core на примере двухядерного процессора Presler, выполненного по 65-нанометровому технологическому процессу, каждое ядро которого имеет собственный кэш второго уровня L2 объемом 2 Мбайт. Он поддерживает 64-битное расширение памяти Intel EM64T и технологию Executable Disable Bit, с отключенной на аппаратном уровне технологии Hyper-Threading.

В отличие от двухядерных процессоров Pentium Extreme Edition и Pentium D в процессоре Presler два ядра размещены на разных кристаллах, то есть это будут два физически разделенных кристалла в одной упаковке в соответствии с рисунком 2. Такой подход к организации двухядерной архитектуры предостав-

ляет достаточно гибкие возможности для отбраковки кристаллов, позволяя увеличить долю выхода годных двухядерных процессоров, а к тому же (и скорее всего, в этом главная причина) это позволяет одновременно с двухядерным процессором Presler выпускать и одноядерный аналог, причем кристаллы для обоих процессоров могут «нарезаться» из одних и тех же пластин [3].

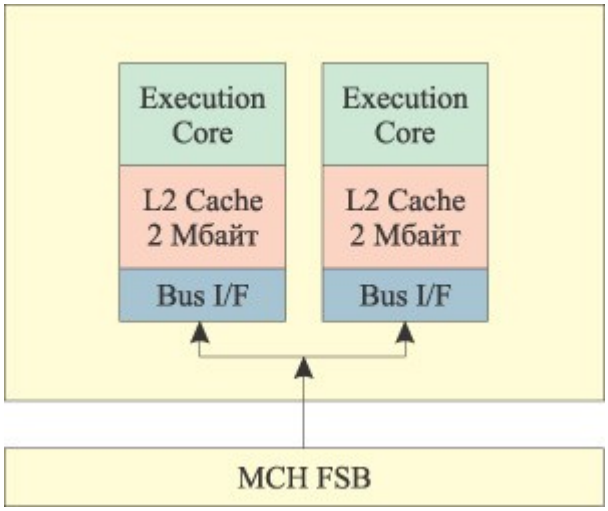


Рисунок 2 - Структурная схема процессора Presler

Микроархитектура Intel Core, реализованная в процессорах Intel позволяет улучшить соотношение производительности и энергопотребления. Характеристики новой микроархитектуры Intel® Core™:

Технология Intel® Wide Dynamic Execution позволяет обрабатывать больше команд за такт процессора, повышая эффективность выполнения приложений и сокращая энергопотребление. Каждое ядро процессора, поддерживающего эту технологию, может выполнять до четырех инструкций одновременно, используя эффективный конвейер из 14 стадий.

Технология Intel® Intelligent Power Capability делает энергопотребление более низким, активируя отдельные логические подсистемы только по мере необходимости.

Технология Intel® Advanced Smart Cache включает совместно используемую кэш-память 2-го уровня, которая снижает энергопотребление, сводя к минимуму обмен данными с памятью, и повышает производительность, позволяя одному из ядер процессора использовать всю кэш-память при бездействии другого ядра.

Технология Intel® Smart Memory Access повышает производительность системы, сокращая время отклика памяти и оптимизируя, таким образом, использование пропускной способности подсистемы памяти.

Технология Intel® Advanced Digital Media Boost позволяет обрабатывать все 128-разрядные команды SSE, SSE2 и SSE3, широко используемые в мультимедийных и графических приложениях, за один такт, что удваивает скорость их выполнения [4].

2.2 Многоядерная технология AMD

Имеющаяся процессорная архитектура AMD позволила интегрировать на тот же кристалл второе ядро при переходе на технологический процесс 90 нм. В процессорах архитектуры AMD64 с двумя ядрами дублированию подвергнуто само вычислительное ядро и кэш-память, в то время как контроллер памяти и контроллер HyperTransport остаются в двухядерных процессорах в неизменном виде. В этой связи ключевое значение начинает играть блок System Request Interface (SRI), на который возлагается обязанность арбитража потоков команд и данных между двумя ядрами. Процессорные ядра взаимодействуют с интегрированным контроллером памяти и контроллером шины HyperTransport через Crossbar Switch, который, по сути, является арбитром шин контроллера памяти и HyperTransport в соответствии с рисунком 3.

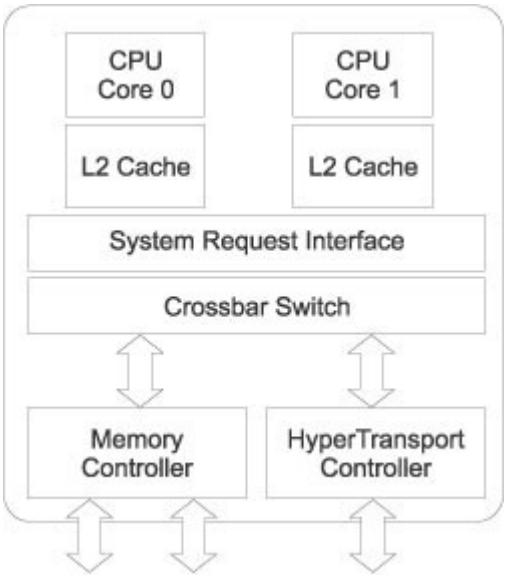


Рисунок 3 - Архитектура процессора AMD Athlon 64 X2 Dual-Core

Особо стоит отметить тот факт, что хотя каждое ядро имеет собственный кэш L2, для него доступны данные и «соседнего» кэша — второго уровня. При этом процессор поддерживает когерентность хранимой в этой «быстрой памяти» информации. Такой подход исключает повторное обращение к оперативной памяти за данными, уже загруженными в кэш одного из ядер, что позволяет уменьшить время ожидания и снизить нагрузку на шину памяти. Кстати говоря, наличие интегрированного на кристалле контроллера памяти также весьма удачно укладывается в концепцию двухъядерности. Ведь в этом случае ядра общаются прямо с контроллером памяти, не имея посредника в лице контроллера системной шины. В своей же практической реализации новые двухъядерные процессоры компании Athlon 64 X2 Dual-Core изготавливаются на основе ядер Toledo и Manchester, для которых одноядерными аналогами являются соответственно ядра San Diego и Venice. Это, в свою очередь, означает, что процессоры выполнены по 90-нанометровому технологическому процессу с применением технологии SOI (Silicon On Insulator) и обладают поддержкой набора инструкций SSE3, а также усовершенствованным контроллером памяти, поддерживающим работу

модулей DDR SDRAM PC1600/2100/2700/3200, в том числе и в двухканальном режиме. Различие ядер Toledo и Manchester, как и San Diego и Venice, состоит в размере кэша второго уровня. Так, в первом случае кэш L2 составляет 2S1024 Мбайт, а во втором — 2S512 Мбайт. При этом тактовая частота процессоров этой линейки равна 2400, 2200 и 2000 МГц, т.е. при переходе на двухъядерную архитектуру не пришлось жертвовать скоростью объединяемых в тандем ядер.

Двухядерные процессоры семейства AMD Athlon 64 X2 Dual-Core, как и семейство Intel Pentium D, ориентированы на использование в графических станциях для работы с 3D-графикой и на универсальные рабочие станции для пользования офисными приложениями, приложениями создания контента, обработке цифровых фотографий и т.д.

Таким образом, как утверждают многие эксперты, в течение ближайших двух лет почти все микропроцессоры, устанавливаемые в настольных ПК, рабочих станциях и серверах, будут иметь два ядра или даже больше. И надо сказать, что основания для таких прогнозов весьма весомые. Производительность двухядерного процессора может быть почти в два раза выше, чем у одноядерного, а его стоимость будет ниже, чем у двух процессоров, имеющих по одному ядру. При размещении двух процессоров на одном кристалле скорость обмена информацией между ними возрастает, а совместное использование кэш-памяти может еще более повысить эффективность обработки данных. Кроме того, двухядерные процессоры занимают меньше места, потребляют меньше энергии и рассеивают меньше тепла, нежели отдельные процессоры. Добавим еще, что, по имеющейся информации, процессоры на базе нескольких ядер хорошо подходят для обработки транзакций, а также для обслуживания баз данных и научных применений [5,8].

3 Особенности проектирования и написания многопоточных программ

Активное внедрение многоядерных систем подразумевает существенное изменение стиля программирования: разработчики будут вынуждены использовать параллельные потоки, порождение и обработку асинхронных событий и др. Иными словами, новая аппаратная архитектура требует смены программной парадигмы — перехода от последовательного стиля программирования к параллельному. Сегодня только небольшая часть программного обеспечения может эффективно выполняться на многоядерных процессорах, что подтверждают результаты тестов — синтетических и предназначенных для конкретных классов приложений (см., например, www.3dnews.ru/cpu/dualcore-cpu/index03.htm). Реальный рост производительности дают лишь программы, оптимизированные под многопоточность, такие как Adobe Premiere Pro 1.5, 3DMax и др. Остаются актуальными задачи разработки и внедрения драйверов устройств, поддерживающих многопоточность [6].

При переходе с одноядерных процессоров на многоядерные приходится принимать во внимание проблему последовательного выполнения. Что это означает применительно к многоядерной системе? В ней выполнение считается последовательным, если в какой-то момент одно или более ядер не могут выполнять код одновременно с другими ядрами. Такая ситуация может возникнуть по разным причинам [1]: блокировка при доступе к ресурсам, необходимость синхронизации процессов или потоков на ядрах, поддержка когерентности кэш-памяти, неравномерность загрузки.

Блокировки возникают из-за невозможности (например, в момент «сбора мусора») одновременного доступа приложений на разных ядрах к жесткому диску, к устройствам ввода/вывода или данным приложений. Очень часто параллельные процессы, выполняемые на разных ядрах, нужно синхронизировать в определенные моменты. Например, приложение на одном из ядер должно использовать промежуточные данные, которые получает приложение (поток, процесс) на другом ядре. Первое приложение не может продолжить работу до получения этих данных, то есть находится в состоянии ожидания. В такой ситуации неизбежны накладные расходы на синхронизацию приложений (процессов, потоков), выполняемых на разных ядрах. В свою очередь, это обусловливает снижение эффективности параллельной работы, что находит отражение в сетевом законе Амдала. Возникает необходимость в поддержке когерентности (согласованности) кэш-памяти для всех ядер при использовании разделяемой памяти.

Можно упомянуть об исследованиях Intel, посвященных динамическому регулированию интенсивности выполнения инструкций (energy per instruction, EPI) в зависимости от степени параллелизма реализации программного обеспечения [2]. Специалисты корпорации опытным путем показали эффективность регулирования тактовой частоты асимметричной многопроцессорной системы в зависимости от активности вычислительных ядер.

3.1 Инструментальные средства многоядерных систем

3.1.1 Компиляторы

Чтобы получить максимальную выгоду от использования многоядерной архитектуры, требуется поддержка на уровне компилятора. Так, в 2005 году Intel выпустила версию 9.0 компилятора языков C++ и Фортран для платформ Linux и Windows. Этот компилятор позволяет эффективно использовать возможности технологии Hyper-Threading и многоядерных процессоров. Он поддерживает возможность автопараллелизма, то есть автоматического обнаружения в приложениях возможности создания множества параллельных потоков с поддержкой спецификации OpenMP 2.5, а также способен генерировать как 32-битный, так и 64-битный программный код с поддержкой наборов инструкций SSE, SSE2 и SSE3. Однако компилятор проверяет производителя процессора (используя инструкцию CPUID) и возвращает ошибку, если в системе установлен не-Intel процессор.

Благодаря поддержке стандарта OpenMP компилятор Microsoft Visual C++ 2005 обеспечивает параллельную многопоточную обработку. Для этого требуется либо указать параметр компилятора «/openmp», либо установить в конфигурации флаг «OpenMP Support». С ноября 2005 года компилятор gcc для языков Си, C++ и Фортран 95 поддерживает OpenMP с помощью опции «-fopenmp». Следует упомянуть и набор компиляторов EKOPath компании PathScale, предназначенных для 64-разрядных систем на базе Linux (AMD64 и EM64T).

3.1.2 Программные отладчики

Отладка многопоточных приложений – задача трудоемкая. При аварийном завершении программы зачастую требуется проанализировать стек вызовов функций во всех потоках, но обычный отладчик показывает только стек потока, на котором произошло аварийное завершение программы. Например, стандартные средства gdb плохо приспособлены для отладки многопоточных приложений, поэтому предлагаются специальные версии этого отладчика для конкретных операционных систем: в их ядра включаются дополнительные возможности отладки многопоточных приложений.

Одна из таких реализаций — отладчик компании Etnus TotalView, предназначенный для платформ Linux, Unix и LynxOS. Он поддерживает многопоточность, MPI, OpenMP, языки программирования Си/C++ и Фортран, а также смешанные коды с использованием разных языков программирования.

Полезным средством оптимизации и отладки параллельных программ является пакет Intel Threading Tools. Он обеспечивает диагностику ошибок и анализ производительности многопоточных приложений, использующих модели потоков Win32 и OpenMP. Отладчик позволяет обнаруживать взаимные блокировки (deadlock) и гонки (race condition) между потоками, временные издержки на переключение потоков (context switch), локализовать проблемы на уровне исходного кода, анализировать эффективность способов повышения производительности OpenMP-программ. Инструмент Intel Thread Profiler автоматически обнаружит проблемы с производительностью внутри многопоточного кода, созданного с поддержкой стандарта OpenMP, изолирует фрагменты кода, содержащие ошибки на уровне функции, строки, переменной или стека вызовов, значительно упро-

щает дальнейший анализ и устранение неисправности. Предоставляет возможность классификации ошибок, т.е. все обнаруженные проблемы могут быть распределены по группам, что упрощает расстановку приоритетов при их устранении, а поддержка стандарта OpenMP позволяет опознать все директивы. Утилита определит, достигло ли приложение своей максимальной производительности и в случае, если потенциал еще не исчерпан, предложит предпринять конкретные шаги по оптимизации программы. Потенциальная производительность и быстродействие кода в многопроцессорных системах отображаются в графическом виде. Этот пакет экономит время разработчика, поскольку автоматическое обнаружение ошибок помогает в поиске неверных элементов многопоточного алгоритма.

Утилита Intel Thread Checker основана на фирменной технологии поиска ошибок, которая отслеживает выполнение программы и определяет местоположение ошибок, угрожающих стабильной работе программы на системах с процессорами, использующими Hyper-Threading. Традиционные методы поиска ошибок основаны на исправлении исходного кода, что может занимать несколько месяцев. Intel Thread Checker позволяет существенно сократить время отладки программы, поскольку указывает конкретные строки кода, в которых содержится ошибка. Утилита также классифицирует ошибки по степени опасности и может выводить для анализа информацию из стека. «Состояние гонки» (Race condition) - одна из наиболее распространенных ошибок, встречающихся в многопоточных приложениях. Указанная ошибка возникает в тех случаях, когда два или несколько потоков одновременно обращаются к одной ячейке памяти и вносят изменения в ее содержимое. Зачастую ошибка «Race condition» может привести к непредсказуемым результатам. Продукт Intel Thread Checker обнаруживает конфликты «race conditions» и другие часто встречающиеся ошибки, в том числе образование тупиков и непредвиденные остановки потоков. Списки диагностики содержат подробную информацию обо всех проблемах, обнаруженных средствами Thread Checker. Каждая запись включает в себя описание ошибки, оценку нанесенного ущерба и номер строки исходного кода. Двойной щелчок кнопкой мыши на записи в списке диагностики отобразит фрагмент исходного кода, в котором обнаружена ошибкаПродукт совместим практически со всеми программными интерфейсами Win32 API и библиотеками C runtime library, а также полностью поддерживает стандарт OpenMP.

Intel Thread Checker интегрирован с графическим интерфейсом продукта Intel VTune Performance Analyzer. В процессе работы с продуктом можно обойтись без дополнительных инструментов, кроме того, не возникает необходимости в повторной компиляции кода для его обработки с помощью VTune Performance Analyzer [7].

3.1.3 Аппаратные отладчики

Для работы с виртуальными машинами аппаратный отладчик должен поддерживать ряд специальных функций (в частности, определять, к какой виртуальной машине относятся те или иные процессы и нити). Их обеспечивает, например, TRACE32 компании Lauterbach. Благодаря полной поддержке встроенных аппаратных блоков управления памятью можно одновременно отла-

живать процессы на нескольких виртуальных машинах и даже два варианта одного процесса на разных виртуальных машинах. В частности, Lauterbach объявила о выпуске программного инструментария интегрированной поддержки ядра (kernel awareness) для операционной системы LynxOS-178. Чтобы получить доступ ко всем функциями TRACE32, не нужно изменять прикладные программы или ядро (применять заплаты, перехватчики, инструментальные «довески» и др.). Отлаживается именно то приложение, которое будет действовать в конечном продукте, что очень важно для его сертификации.

Среди других аппаратных отладчиков, поддерживающих работу с многоядерными конфигурациями, назовем Green Hills Probe и SuperTrace компании Green Hills, WindPower ICE компании Wind River, RealView ICE от ARM.

3.2 Стандарты многоядерных систем

При разработке параллельных программ используются специализированные библиотеки и системы параллельного программирования MPICH, PVM, LAM, CHMP и др. Три основных подхода к реализации этих систем различаются методами взаимодействия параллельных задач. Первый подход базируется на концепции обмена сообщениями, второй — на использовании разделяемой памяти, третий опирается на стандарт POSIX и объединяет эти два подхода.

Наиболее известным представителем первой группы является спецификация MPI (Message Passing Interface) для языков Си и Фортран, первый вариант которой появился в 1994 году. MPI обеспечивает примерно 200 функций, охватывает множество компиляторов и операционных систем. Среди наиболее распространенных ее реализаций — библиотека MPICH [10]. Кроме того, предлагаются несколько коммерческих реализаций MPI, например MPI/Pro компании Verari Systems Software. MPI/Pro оптимизирует время работы параллельных приложений и поддерживает их масштабируемость за счет балансировки параметров производительности и использования ресурсов. Verari предлагает версии MPI/Pro для разных операционных систем, в том числе Windows, Linux, Mac OS X, LynxOS, и таких коммуникационных сред, как Gigabit Ethernet, Myrinet и InfiniBand [9].

Ко второй группе относится спецификация OpenMP (Open specifications for Multi-Processing). Ее первая версия (www.openmp.org), которая была выпущена в 1997 году, предназначалась для языка Фортран. К появлению OpenMP «приложили руку» компании IBM, Intel, Sun Microsystems и Hewlett-Packard. В 1998 году были созданы варианты OpenMP для языков Си/C++, а последней является версия 2.5. Поддержка спецификации OpenMP обеспечена во всех компиляторах Intel начиная с шестой версии, в Microsoft Си/C++ начиная с Visual Studio 2005, а также в GCC.

OpenMP — это набор специальных директив компилятору (pragma), библиотечных функций и переменных среды. Наиболее оригинальны директивы компилятору, которые используются для обозначения областей в коде и могут выполняться параллельно. Компилятор, поддерживающий OpenMP, преобразует исходный код и вставляет соответствующие вызовы функций для параллельного выполнения этих областей кода. В третью группу входит спецификация POSIX (Portable Operating System interface for unIX), первое описание которой было пуб-

ликовано в 1986 году (www.pasc.org). Основная спецификация разработана как IEEE 1003.1 и одобрена как международный стандарт ISO/IEC 9945-1:1990. С точки зрения организации параллельных вычислений наибольший интерес представляют три части стандарта — 1003.1a (OS Definition), 1003.1b (Realtime Extensions) и 1003.1c (Threads). В рамках POSIX можно реализовать параллельные вычисления на основе обмена сообщениями (аналогично MPI) или разделяемой памяти (как в OpenMP). Естественно, в POSIX допустима и любая комбинация этих методов. В наибольшей степени стандарту POSIX соответствуют (и соответствующим образом сертифицированы) операционные системы реального времени LynxOS и Integrity.

3.2.1 Поддержка на уровне ОС

Многоядерные процессоры потребуют от операционных систем поддержки разных архитектур многопроцессорной обработки. Компания QNX Software Systems объявила о выпуске комплекта разработчика QNX Momentics Multi-Core Edition. Этот набор инструментов предназначен для создания программного обеспечения и его миграции на многоядерные аппаратные решения нового поколения, в том числе процессоры BCM12xx и BCM14xx компании Broadcom, процессор MPC8641D компании Freescale и многоядерные процессоры Intel. Будут поддерживаться несколько моделей многопроцессорности для многоядерных архитектур: асимметричная AMP (обеспечение полного управления и отказоустойчивости); симметричная SMP (максимальные параллелизм и масштабируемость); «исключительная» BMP (поддержка миграции кода и снижение сложности разработки).

Поддержку многоядерных систем на базе процессоров AMD64, Sun UltraSPARC T1 и Intel обеспечивает ОС Solaris 10. Например, встроенная система виртуализации и защиты информации Solaris Containers позволяет системному администратору организовывать в рамках единой операционной системы несколько виртуальных системных разделов — «зон». Каждой зоне допустимо назначить свой контейнер — набор локализованных системных ресурсов. Контейнеры могут служить основой для управления ресурсами на уровне ядер. Реализованные в Solaris 10 функции так называемого «прогнозируемого самовосстановления» (Predictive Self-Healing) обеспечивают автоматическое определение сбоев в работе ядер и их перевод в пассивный режим без влияния на работу остальных ядер процессора. Поддержка многоядерных систем реализована в некоторых дистрибутивах ОС Linux, например Red Hat Enterprise Linux 4.

3.2.2 Многоуровневая виртуализация

Появление многоядерных процессоров спровоцирует массовое внедрение технологий виртуализации. Назовем некоторые из известных подходов:

ARINC-653 (Avionics Application Software Standard Interface). Стандартный интерфейс, разработанный компанией ARINC в 1997 году, вводит концепцию изолированных разделов на основе универсального программного интерфейса APEX (Application/Executive) между операционной системой и прикладным программным обеспечением. Требования интерфейса определены так, чтобы разрешить приложениям контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов.

В 2003 году принята новая редакция ARINC-653, в которой введена концепция изолированных виртуальных машин. Ее особенностью является жесткое и заранее определенное квантование времени между виртуальными машинами, а целью — обеспечение гарантий того, что не возникнут общие отказы системы. Стандарт ARINC-653 реализован для операционных систем реального времени LynxOS-178, VxWorks, Integrity, CsLeos и др.

User-Mode Linux (UML) для операционной системы Linux в пользовательском режиме самый универсальный эмулятор, позволяющий создавать виртуальное оборудование, которого может и не быть на физическом компьютере. Это весьма удобно для тестирования конфигураций аппаратного обеспечения. UML состоит из набора заплат к ядру Linux, которые позволяют запускать другие операционные системы в консольных окнах, и каждый пользователь может независимо загружать сколько угодно операционных и оконных систем, вплоть до X11. User-Mode Linux допускается применять для устройств с архитектурой IA-32 и PowerPC G5.

Программные среды виртуальных машин. Наиболее популярными из них являются Microsoft Virtual PC и группа программных продуктов VMware. Система виртуальных машин позволяет запускать на компьютере сразу несколько разных операционных систем и переключаться с одной на другую без перезапуска компьютера. На компьютере, работающем под управлением основной (базовой) операционной системы, создаются один или несколько виртуальных компьютеров, на каждом из которых можно запустить «гостевую» ОС.

VMWare Workstation позволяет запустить несколько экземпляров Windows, Linux и NetWare. Реализованы полноценная поддержка сети, переносимость окружений и гибкий подход к работе с окружением. Проект Virtual PC изначально разрабатывала компания Connectix, но в начале 2003 года его купила корпорация Microsoft. К сожалению, после этого Virtual PC лишился поддержки «гостевых» Unix-подобных систем (в том числе, Linux) и был полностью ориентирован на установку Windows-систем на других платформах.

Технология виртуализации Intel. VT, компонент многоядерной технологии поддержки виртуализации на аппаратном уровне [3], обеспечивает поддержку виртуальных машин на уровне процессора с помощью нового режима VMX (Virtual Machine Extensions) и десяти команд — vmptrld, vmptrst, vmclear, vmread, vmwrite, vmcall, vmlauch, vmresume, vmxoff и vmxon. При этом повышаются как надежность и производительность работы приложений, так и уровень общей безопасности.

Архитектура VT поддерживает два класса ПО: монитор виртуальной машины VMM и «гостевое» программное обеспечение. Используются два режима работы — root operation и non-root operation. Как правило, VMM работает в первом режиме, а «гостевые» программы — во втором. Поддержку технологии виртуализации Intel намерены организовать такие производители операционных систем, как RedHat, SuSe и MontaVista. Она будет обеспечена и в других программных средствах виртуализации, например в VMware.

4 Вопросы по теоретической части

4.1 Изложите возможную классификацию архитектур, используя в качестве критерия разделения по группам количество потоков команд и количество потоков данных.

4.2 Перечислите модели памяти, которые могут быть реализованы в различных архитектурах вычислительных систем.

4.3 В чем состоят особенности архитектуры многоядерных процессоров AMD?

4.4 Приведите примеры программного обеспечения, осуществляющего профилирование, таймирование, а также анализ взаимодействия потоков в приложении.

4.5 Что общего и чем отличаются методы подготовки последовательных и параллельных программ? Какие проблемы затрудняют преобразование одних в другие?

4.6 Дайте характеристику библиотеки Pthread языка C, обеспечивающей поддержку многопоточности.

4.7 Какой механизм использован для управления потоками в операционной системе Windows?

5 Причины плохой масштабируемости программ

Основными причинами плохой масштабируемости программ являются алгоритмические ограничения, аппаратные и программные факторы.

Ограничения, вносимые алгоритмом, связаны с невозможностью выделить независимые участки одинаковой вычислительной сложности, которые можно выполнять параллельно. В случае, когда количество процессоров превысит количество этих независимых участков, перестанет расти ускорение и программа станет плохо масштабируемой.

Аппаратные факторы включают в себя ограничения, вносимые особенностями архитектуры и количественными характеристиками устройств.

Программные факторы представляют собой неэффективную программную реализацию алгоритма. Они могут включать в себя, например, неудачный способ организации, размещения и обмена данными, низкую эффективность распараллеливания. Иногда проявление программных факторов тесно связано с аппаратными ограничениями. Для аппаратных и программных факторов действует правило: при скоплении очереди к любому ресурсу (данным, аппаратным средствам) происходит падение производительности. Перенос программы на архитектуру с большим количеством процессоров\ядер может не разрешить проблему или даже ухудшить ситуацию. Рассмотрим более детально возможные проблемы, связанные с аппаратными и программными факторами.

Проблема - очередь запросов к шине. При чтении-записи больших объемов данных шина перестает справляться с их перекачкой. С увеличением числа процессоров проблема усугубляется, так как им требуется больше данных в единицу времени.

Проблема - очередь к критическому ресурсу. Зависание процессов (lock) может произойти при некорректном использовании механизмов синхронизации, например, спин-блокировок. Большое количество критических секций также способствует организации очереди к ресурсу. Использование механизмов синхронизации, требующих больших накладных расходов вместо более оптимальных для данной задачи может повлечь падение ускорения программы.

Проблема – очередь к разделяемому ресурсу. При чтении\записи данных на диск, на сетевую карту, принтер и любое другое аппаратное обеспечение, тактовая частота которого ниже тактовой частоты процессора, возникает очередь запросов чтения\записи.

Проблема - очередь запросов к процессору. Неоправданно большое количество потоков в пересчете на процессор влечет за собой частое переключение контекста, при котором происходит запоминание содержимого регистров и их сброс, восстановление содержимого регистров переключаемого потока и пр., а это накладные расходы, влияющие на ускорение программы.

Проблема - очередь запросов в пуле потоков к потоку-менеджеру. Если размер пула меньше оптимального, потоки встанут в очередь, если размер больше оптимального, то поток-менеджер не успеет обработать возрастающее число запросов.

6 Лабораторная работа № 1 - Влияние пропускной способности шины данных на масштабируемость программ

В данной лабораторной работе делается упор на одну из возможных проблем при масштабировании, а именно, на недостаточную по сравнению с процессором, пропускную способность шины данных, которая может привести к ухудшению прироста производительности программы при увеличении числа процессоров и возросшего количества запросов к памяти.

Конечно, пропускная способность шины данных является ограниченной величиной. И, хотя в настоящее время, скорости работы оперативной памяти и шины данных очень велики, они все же не поспевают за скоростью работы процессора. Таким образом, иногда могут складываться ситуации, в которых шина данных «не справляется» с быстро поступающими данными и тем самым заставляет процессор простаивать.

Рассмотрим ситуацию с гипотетическим многопоточным приложением, которое выполняет интенсивную работу с памятью во многих потоках сразу. На системах с одним процессором это приведет к замедлению работы по сравнению с шиной с «бесконечной» пропускной способностью. В многопроцессорных системах с общей шиной данных, которые призваны ускорить работу многопоточных приложений, может произойти ситуация, когда шина данных не будет успевать обрабатывать увеличенный поток запросов от большего количества процессоров. В этом случае отношение прироста производительности к затраченным на это средствам (а вертикальное масштабирование, как известно, не является самым дешевым способом увеличить производительность) может оказаться неудовлетворительным. Уменьшить влияние этого эффекта можно 2 способами:

а) аппаратный – физическое повышение пропускной способности шины, использование многоканальной памяти и др.

б) программный – оптимизация количества запросов в единицу времени за счет изменения алгоритма; изменение схемы распараллеливания и др.

Цель работы: оценить влияние пропускной способности шины на масштабируемость программ

Необходимое программное обеспечение: операционная система Microsoft Windows 2000, среда разработки Microsoft Visual Studio 6.0 и выше, и программа, реализующая поворот картинок в памяти.

6.1 Формулировка задачи

Для анализа проблемы предлагается:

1 использовать программу, намеренно пытающуюся «забить» шину данных и в несколько потоков активно работать с памятью;

2 оценить теоретическое ускорение и масштабируемость по закону Амдаля;

3 провести серию тестов параллельной и последовательной версии программы на разных конфигурациях вычислительных систем (одно-, двух- и четырехпроцессорные, с разной пропускной способностью шины данных) для выявления проблем масштабируемости;

4 по результатам тестов заполнить таблицы и построить графики ускорения работы программы при увеличении числа процессоров. Оценить практическое ускорение и масштабируемость;

Для демонстрации проблемы разработана программа, осуществляющая повороты BMP-файлов, целиком хранящихся в памяти. Схема работы программы отражена на рисунке 4.

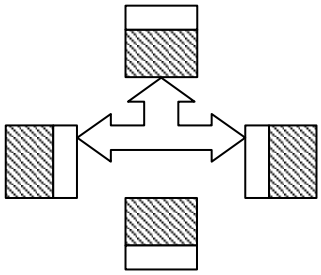


Рисунок 4 – Схема многопоточного поворота картинок

6.2 Реализация последовательного приложения[1]

Для реализации данной задачи в операционных системах Microsoft Windows существует достаточное количество API-функций, позволяющих создавать многопоточные приложения, работать с форматом файлов BMP, а также с высокой точностью оценивать производительность, как полного времени выполнения задания, так и время работы отдельных потоков.

Для работы с форматом BMP необходимо использовать стандартные средства GDI: CreateDC (создает графический контекст), CreateCompatibleDC (создает совместимый с данным контекст), SelectObject (выбирает графический объект в контекст), BitBlt (быстрое копирование битов с контекста на контекст), GetDIBits (получение массивов данных BMP). Программа написана на Visual C++ 6.0 с использованием MFC. Весь полезный код находится в файле bmp_parallDlg.cpp. Для подсчета полного времени работы лучше использовать функции QueryPerformanceFrequency и QueryPerformanceCounter, которые позволяют получать время с точностью до 100 наносекунд. Работа напрямую с API значительно ускоряет работу. Так, для сравнения, код, написанный на Borland VCL выполнялся в 30-40 раз медленнее (!).

В главной функции – doRotation данной функции делается снэпшот экрана (а именно BMP размером 6000 * 6000 пикселей с глубиной цвета 32 бита), затем из полученного BMP-изображения извлекаются биты в массив с элементами по 4 байта, который хранится в динамически распределяемой памяти в качестве исходного массива. Поворот по-, против-,

[1] Пример подготовлен совместно с А. Ларченко

вокруг часовой стрелки осуществляется функцией threadProc. После выполнения работы выдаются результаты - время в миллисекундах.

6.3 Инструменты анализа производительности приложения

Для анализа качества распараллеливания, утилизации ресурсов и оптимизации программы будем использовать инструменты: Microsoft Performance Monitor, Intel Thread Profiler, Intel VTune Performance Analyzer. Первичный инструментированный анализ программы выполним с помощью Performance Monitor.

Performance Monitor – встроенный в Windows инструмент для оценки производительности приложений и системы в целом. Особенно полезными представляются два объекта: Process и Memory. Объект Process собирает информацию обо всех работающих процессах, будь то системные процессы, пользовательские процессы или службы Windows. Объект Memory собирает данные, описывающие компоненты подсистемы управления памятью ОС, включая файловый кэш, физическую память, а также несколько выгружаемых и невыгружаемых пулов памяти, которые Windows NT использует для системных процессов. Более подробно о том, как работать с этим инструментом в целях сбора информации о компьютере, можно узнать из документации Microsoft и специальной литературы.

Intel Thread Profiler – утилита для оптимизации производительности многопоточных приложений Win32 и OpenMP. Контролируя работу приложения, она фиксирует все возникающие проблемы, включая рассинхронизацию потоков и временные издержки на переключение между потоками. Полученные данные отображаются в графическом виде, который позволяет быстро идентифицировать фрагменты исходных текстов, нуждающиеся в доработке.

Intel VTune Performance Analyzer предназначен для поиска «узких мест» в 32- и 64-битных -приложениях Windows, а также в Java-приложениях для систем с архитектурой IA-32. Он обеспечивает сбор и анализ данных об интегральной производительности процессов и приложений, а также их отдельных составляющих: функций, модулей и инструкций.

6.4 Реализация параллельного приложения

Для создания потоков в Windows используется функция CreateThread, передающая ряд параметров, важнейшими из которых являются функция, в которой выполняется код потока и указатель на данные потока. Поток можно создавать приостановленным или сразу запущенным.

Для подсчета времени работы отдельных потоков используется функция GetThreadTimes, которая возвращает время выполнения потока в режиме ядра и в режиме пользователя. Главная функция – doRotation, которой передается булева переменная – выбор пользователем параллельной или последовательной версии. Интерфейс программы (рисунок 5) прост: кнопка [Parall] запускает распараллеленную версию поворотов, кнопка [Consec] – последовательную. Кроме этого, для параллельной вер-

сии реализована оценка времени работы потоков без учета накладных расходов на их создание.

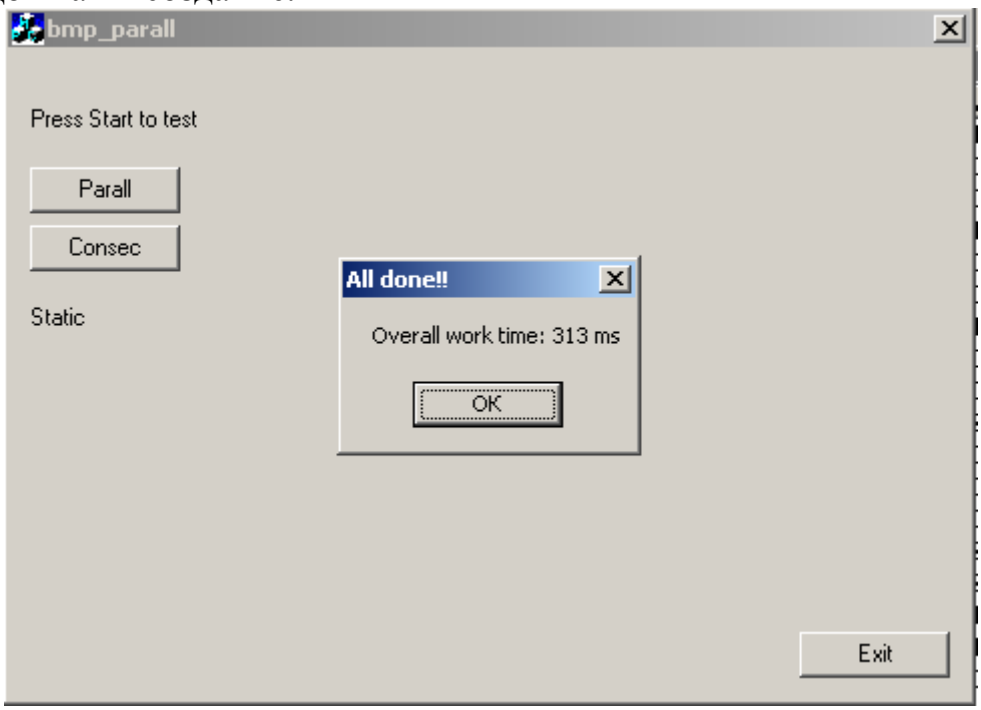


Рисунок 5 – Скриншот работы программы

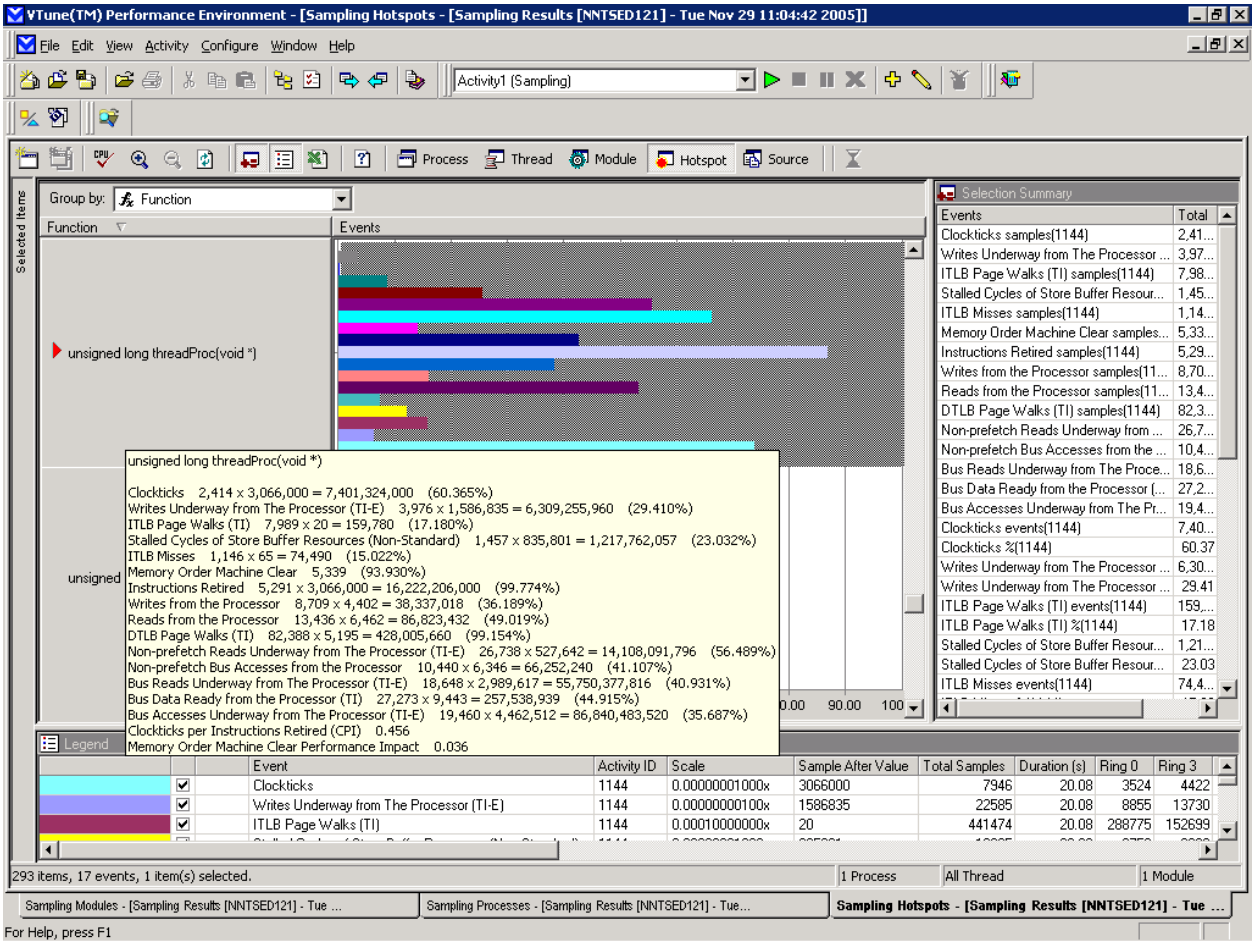


Рисунок 6 – Анализ HotSpot-функции treadProc

Intel VTune Analyzer позволяет выявить функции, отбирающие большую часть времени, так называемые HotSpot-функции. Для нашей программы это функция, непосредственно производящая поворот картинок в памяти. На рисунке 6 представлен анализ ее работы. Как видно из приведенных тестов, основным узким местом программы является процедура threadProc, в которой производятся все действия по повороту картинки – она забирает 60% процессорного времени.

Пример кода HotSpot – функции threadProc:

```
/* функция потока: собственно, поворот bmp-файла */
DWORD WINAPI threadProc(LPVOID pData)
{       MYDATA data = *(PMYDATA) pData; // получаем инфу о текущем потоке
        long index = data.index; // индекс потока в массиве
        long rotation = data.rotation; // тип вращения
        pBGR src = data.src; // массив-источник
        pBGR dst = data.dst; // исходный и конечный массивы точек
        long sz = data.size; // размер bmp-файла (лучше не использовать глобальную константу BMP_SIZE)
        long x, y; // координаты
        register BGR bgr; // промежуточная переменная
        // ходим по bmp-файлу
        for (y = 0; y < sz; y++) {
                for (x = 0; x < sz; x++) {
                        switch (rotation) {
                                case ROT_CW:
                                        bgr = src[y * sz + sz - x - 1];
                                        dst[x * sz + y] = bgr;
                                        break;
                                case ROT_CCW:
                                        bgr = src[(sz - y - 1) * sz + x];
                                        dst[x * sz + y] = bgr;
                                        break;
                                case ROT_2W:
                                        bgr = src[y * sz + x];
                                        dst[(sz - y - 1) * sz + x] = bgr;
                                        break;
                        }               }       }
        delete [] dst; // удаляем массив, в который переносили информацию
        return 0; }
```

6.5 Предварительная оценка ускорения при распараллеливании HotSpot функции

Пусть для решения некоторой задачи с помощью последовательной программы, выполняемой на одном процессоре, нужно время Ttotal, а с

помощью параллельной программы, выполняемой на p процессорах — Tparal. Тогда ускорение S (speedup) параллельной программы определяется как S= Ttotal/Tparal.

Теоретическую оценку ускорения проведем по закону Амдаля:

$$\frac{Ttotal}{Tparal} = \frac{1}{f + (1 - f)/p}, \quad (1)$$

где f – доля последовательного кода;
p – количество процессоров;
Ttotal – время работы последовательной программы;
Tparal – время работы параллельной программы.

Вернемся к рисунку 5. HotSpot-функция threadProc забирает 7401324000 такта, что составляет 60% от общего количества тактов процессора. Поэтому в функции threadProc сосредоточено 60% потенциально параллельного времени. При хорошем распараллеливании, на 2 процессорах она будет выполняться 30 % от общего времени. Следовательно, общее время распараллеленной версии составит 70% от последовательной. Если последовательная программа выполняется 1281 мс, то параллельная должна закончить свою работу за 896.7 мс. Тестовые данные, приведенные в следующем разделе, дают другую картину.

6.6 Практическая оценка ускорения работы программы

Сначала, о конфигурации тестов - программа тестировалась с 2 и 6 потоками (количество потоков задается константой THREADS_COUNT).

Таблица 1 – Mobile, 2 потока

Запуски	Параллельная версия	Чистое поточное время	Последовательная версия
Запуск 1	1122	930	1126
Запуск 2	1134	935	1122
Запуск 3	1125	926	1122
Запуск 4	1111	922	1130
Запуск 5	1118	925	1126
Среднее	1122	927.6	*1134.8
Ускорение	1,01		

Использовались компьютеры с процессором Intel Pentium 4 (M), 1.7 GHz с памятью 1 Gb под управлением Microsoft Windows XP Professional 2002 и dual Xeon 3.06 GHz, 2 Gb RAM под управлением ОС Microsoft Windows Server 2003. Результаты тестов приведены в таблицах 1 – 4 с указанием конфигураций и количества потоков. Теперь постараемся их интерпретировать. Во-первых, полученное ускорение меньше чем теоретическое, во-вторых при переходе на параллельную версию оно очень незначительно повышается, а переход на двухядерную машину также не дает улучшения.

Таблица 2 – Xeon, 2 потока

Запуски	Параллельная версия	Чистое поточное время	Последовательная версия
Запуск 1	1075	879	1304
Запуск 2	1086	890	1266
Запуск 3	1094	895	1273
Запуск 4	1089	893	1269
Запуск 5	1088	832	1293
Среднее	1086,4	877.8	1281
Ускорение	1,18	Ускорение относительно *	1134.8/1086,4= 1.04

Таблица 3 – Mobile, 6 потоков

Запуски	Параллельная версия	Чистое поточное время	Последовательная версия
Запуск 1	6024	783	3269
Запуск 2	3432	752	3208
Запуск 3	3180	769	3243
Запуск 4	4244	1054	3201
Запуск 5	3200	870	3198
Среднее	4016	845.6	**3223.8
Ускорение	0,80		

Таблица 4 – Xeon, 6 потоков

Запуски	Параллельная версия	Чистое поточное время	Последовательная версия
Запуск 1	2915	1216	3405
Запуск 2	2586	1322	3417
Запуск 3	2925	1213	3404
Запуск 4	2677	918	3409
Запуск 5	2527	826	3413
Среднее	2726	1099	3409,6
Ускорение	1,25	Ускорение относительно **	3223.8/2726= 1.18

Сделан вывод, что выдвинутое положение о плохой масштабируемости данной программы верно (значит, шина была достаточно загружена), т. к. прирост относительного времени составляет всего несколько процентов. Попробуем увеличить количество потоков до 6 и протестировать (таблица 6). Видно, что при сильной загрузке шины, когда 6 потоков интенсивно перемещают данные из между разными областями памяти, производительность приложения на однопроцессорной машине ухудшается - ускорение падает до 0,8027. Переход на двухядерную архитектуру также не дает пропорционального прироста производительности. С помощью инструмента Intel Thread Profiler попробуем оценить качество распараллеливания. Перекомпилируем программу, включив генерацию символьной информации (-Zi or /Zi) и добавив в настройки проекта в Linker -> Command Line опцию /fixed:no. Скомпилируем приложение с thread-safe run-time библиотеками, используя опцию -MD или –MT. Запустим Intel Thread Profiler и созда-

дим новый проект. Уменьшим для тестового режима размер bmp-файла до 300 пикселей и для 2 потоков получим приблизительно такие результаты:

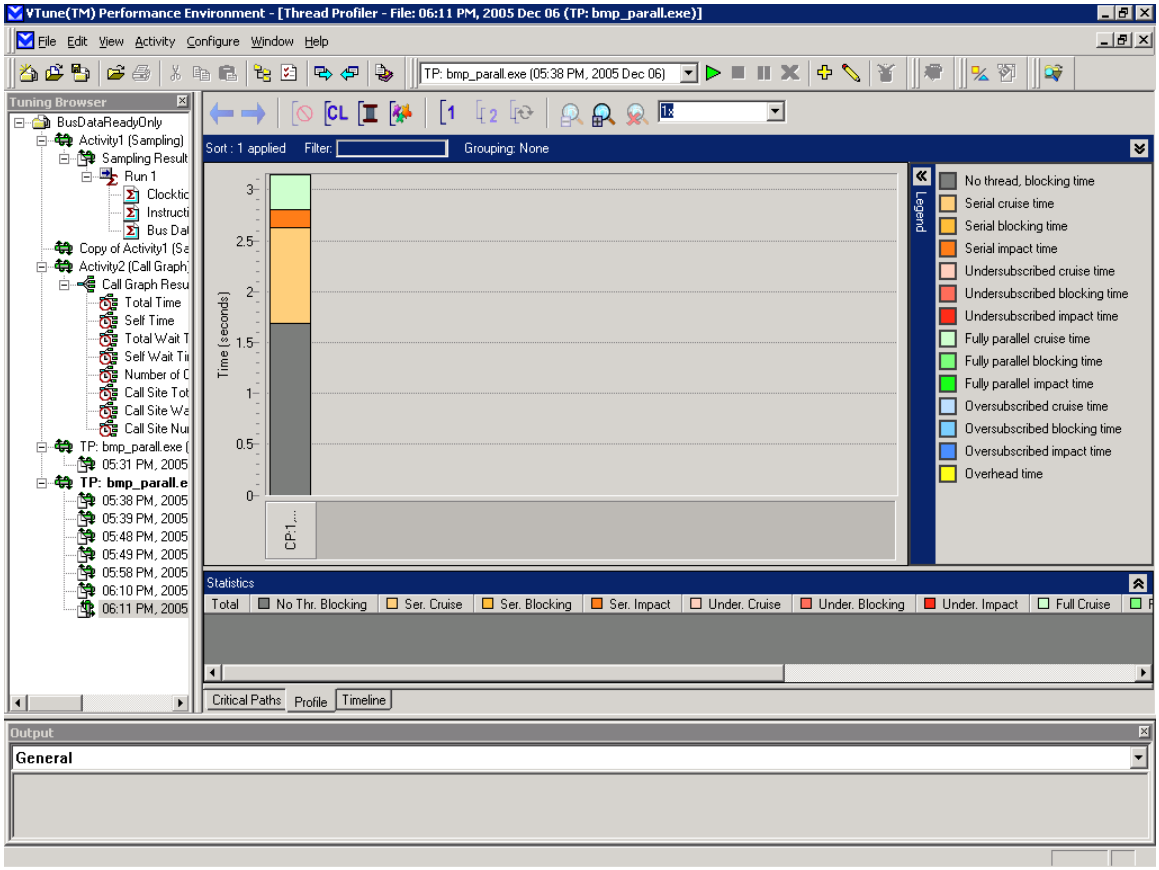


Рисунок 7 – Анализ работы 2-х поточной программы с помощью Thread Profiler

Анализируя рисунок 7, отметим, что первые 1.7 сек. программа ожидает ввода пользователя (no thread, blocking time) - это время предлагается исключить из оценки, т.к. время отклика пользователя непостоянная величина. Следующие 0.9 сек тратится на создание потоков без их запуска и только после этого программа переходит в параллельный режим. Видно, что доля параллельных вычислений небольшая (зеленый прямоугольник) – 0.4 сек. Вывод – функцию threadProc нужно оптимизировать.

6.7 Оптимизация программы для повышения доли параллельного кода

Для оптимизации HotSpot-функции threadProc предлагается сделать следующие действия:

1 выделить память под приемник пикселей в стеке потока, переведя операцию dst= new BGR[BMP_SIZE*BMP_SIZE] из основной программы в функцию потока;

2 вынести оператор switch за пределы цикла;

3 перенести вызовы функций получения контекста экрана CreateDC, создания совместимого контекста экрана CreateCompatibleDC, создания картинки, совместимой с контекстом экрана CreateCompatibleBitmap и получения пикселей картинки GetBMPbuffer в функцию потока.

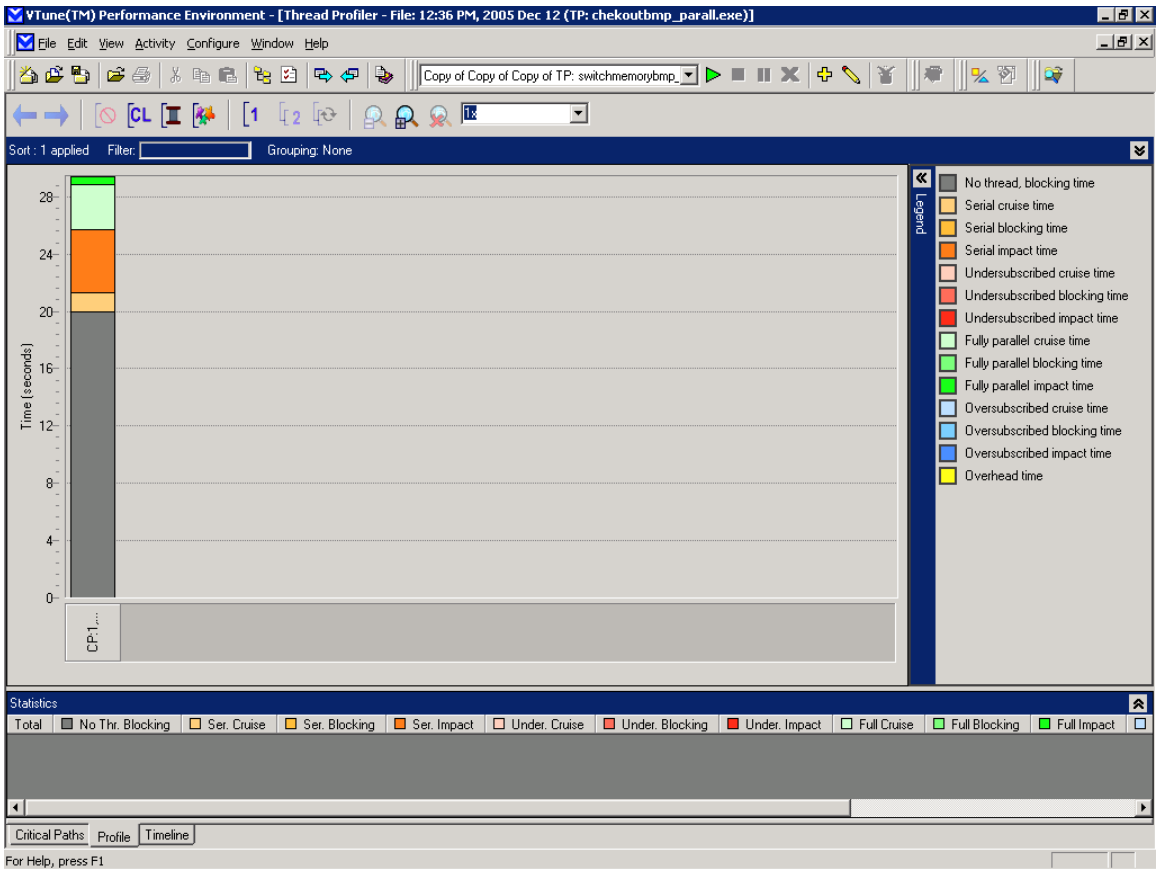


Рисунок 8 – Оценка параллельной доли.

С помощью ThreadProfiler проведем диагностику качества распараллеливания оптимизированного варианта приложения – рисунок 8.

Как видно, достаточно простая оптимизация позволила улучшить результат с 1.18 секунд до 1.36 секунд. В таблице 5 приведены результаты теста оптимизированного проекта.

Таблица 5 – Xeon, 2 потока

Запуски	Параллельная версия	Последовательная версия
Запуск 1	893	1233
Запуск 2	884	1291
Запуск 3	879	1235
Запуск 4	921	1174
Запуск 5	896	1167
Среднее	894.6	1220
Ускорение	1.36	

6.8 Оценка ускорения и масштабируемости после оптимизации

Очевидно, что лучше произвести оценку именно доли последовательного кода, а не параллельного, т.к. она будет постоянна. Для оценки доли работы последовательного кода предлагается установить функции-счетчики QueryPerformanceFrequency и QueryPerformanceCounter, которые позволяют получать время с точностью до 100 наносекунд. Выявленная таким образом доля последовательного кода f для оптимизированного варианта составляет 0.465.

При количестве процессоров, равном 2, максимальное ускорение определяется по формуле (1):

$$\frac{Ttotal}{Tparal} = \frac{1}{0.465 + (1 - 0.465)/2} \approx 1.365 .$$

Теоретически рассчитанное ускорение программы на четырехпроцессорной системе составит 1.67. При идеальном распараллеливании и бесконечном количестве процессоров максимальное ускорение будет приблизительно равно 2.14.

Таблица 6 – 4P Xeon, 3 потока

Запуски	Параллельная версия	Последовательная версия
Запуск 1	783	1138
Запуск 2	767	1118
Запуск 3	701	1127
Запуск 4	712	1116
Запуск 5	794	1111
Среднее	751.4	1122
Ускорение	1.49	

Для оценки масштабируемости программы проведем запуск 3-х и 4-х поточной версии на 4-х процессорном Xeon, с установленной ОС Windows 2003 Server.

Таблица 7 – 4P Xeon, 4 потока

Запуски	Параллельная версия	Последовательная версия
Запуск 1	783	1178
Запуск 2	767	1118
Запуск 3	701	1167
Запуск 4	712	1117
Запуск 5	794	1127
Среднее	751.4	1141.4
Ускорение	1.52	

Таким образом, по графикам рисунка 9 видно, что практическая оценка масштабируемости приближается к теоретической и является нелинейной. Более того, при увеличении числа процессоров с 4 до 8 масштабируемость программы будет ухудшаться из-за предельной нагрузки шины.

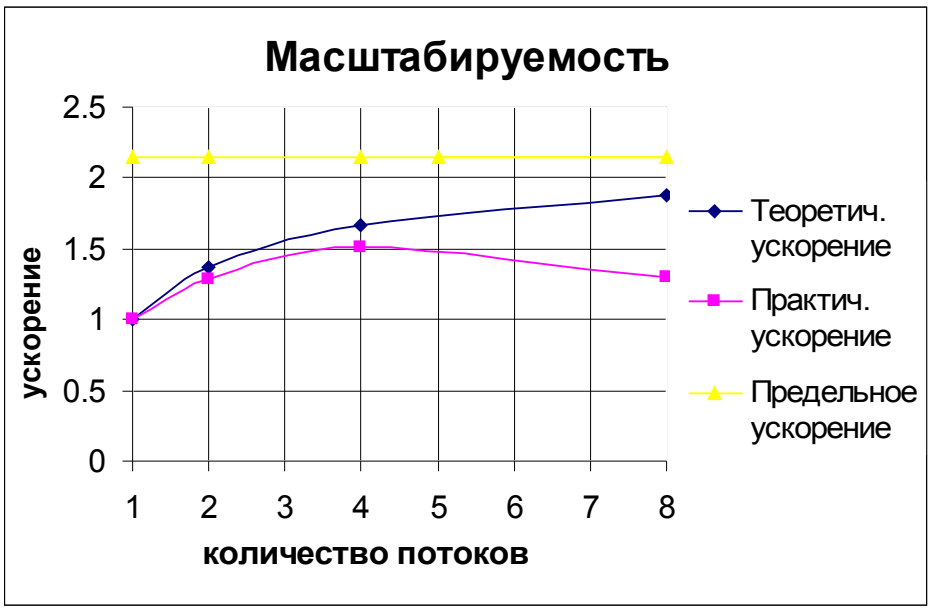


Рисунок 9 – Оценка масштабируемости приложения.

6.9 Инструментированный анализ загрузки шины

Чтобы проанализировать причины недостаточного увеличения производительности при переходе на лучшую архитектуру, используем инструмент Intel VTune Analyzer. Так как мы намеренно загружали шину данных, проведем мониторинг событий на шине данных и памяти. Запускаем VTune Analyzer 7.1 или более позднюю версию. Чтобы сконфигурировать активность для .NET проекта находим Create New Project, выбираем Sampling Wizard и делаем OK. Выбираем .NET Profiling. В следующем диалоговом окне находим тип приложения Executable. Для анализа загруженности шины выберем события шины и процессора.

События шины:

Bus Data Ready (This Processor) - Количество тактов шины, потраченных на транспортирование данных к ядру процессора, включая полные и частичные операции чтения-записи.

Bus Reads Underway (This Processor) - Накопленная сумма времен всех транзакций чтения. При делении на Reads from the Processor получим Bus read request latency (Задержку шины на запросы чтения).

Reads (This Processor) - Число всех запросов чтения, которые были предназначены для Очереди ввода-вывода от ядра процессора, включая предсказанные транзакции.

Writes Underway (This Processor) - Накопленная сумма времен всех транзакций записи. При делении на Reads from the Processor получим Bus write request latency (Задержку шины на операции записи).

Writes (This Processor) - Число всех запросов записи, которые были предназначены для Очереди ввода-вывода от ядра процессора.

События процессора:

Clockticks – количество тактов процессора

Instruction Retired – количество выполненных инструкций.

Результаты сбора данных представлены на рисунке 10.

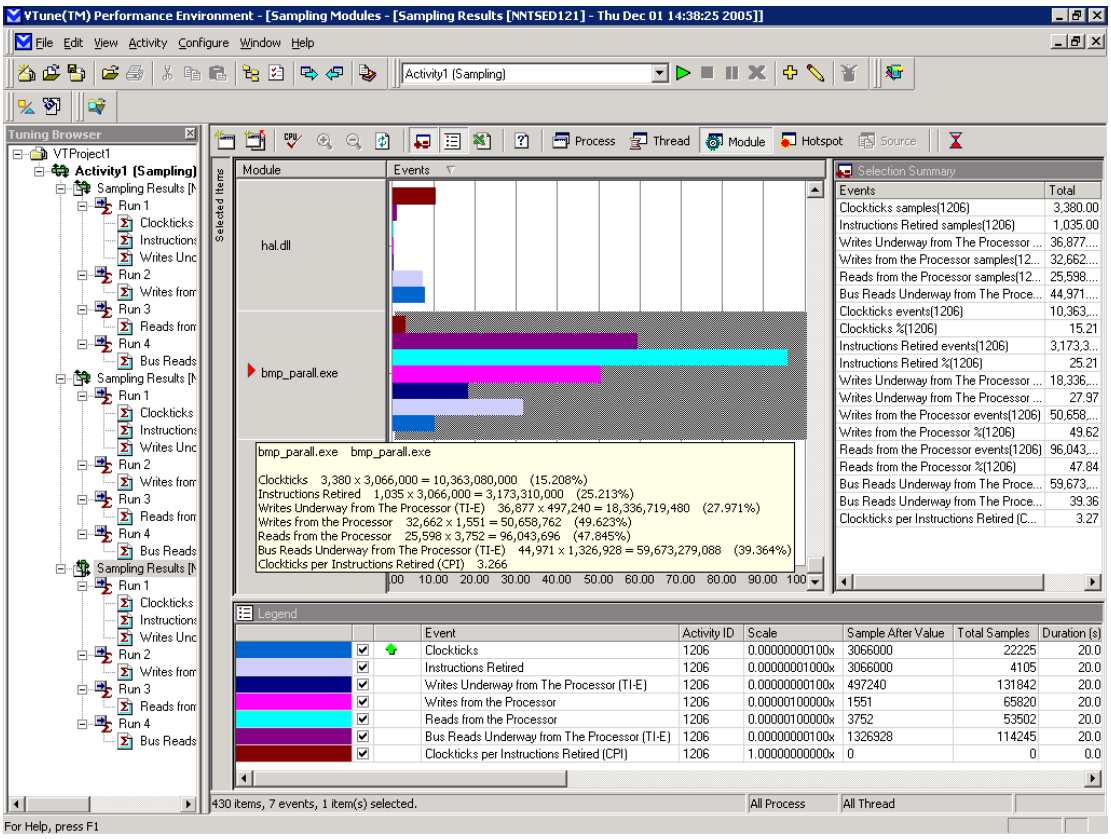


Рисунок 10 – Мониторирование событий шины для последовательного варианта запуска приложения на Xeon

Расчет задержки шины на запросы чтения\записи (Bus Read/Write request latency) проведем по формулам (2):

Bus Read request latency=Bus Reads Underway\Reads from Processor (2)

Bus Write request latency = Bus Writes Underway\Writes from Processor.

Рабочая задержка шины на запрос чтения\записи для Intel P4 лежит в пределах 200-300 тактов. При получении больших значений становится ясно, что работу с памятью нужно оптимизировать. Анализ загрузки цины проведем по формулам (3).

Расчет теоретической пропускной способности шины:

Theoretical bandwidth = Bus frequency * 8.

Расчет базовой частоты шины:

Bus base frequency = Bus frequency / 4. (3)

Расчет реальной загрузки шины:

Bus bandwidth = Bus Data Ready*Bus ratio*Theoretical bandwidth/ Clockticks

Bus ratio real=BusDataReady/(Clockticks*Bus base Freq/ProcessorFreq).

Таблица 8 – Расчет загрузки шины для последовательной версии

Событие	Значение	Событие	Значение
Bus Data Ready	33490624	Bus Write request latency	361.97
Clockticks	1092308000	Processor frequency	3000 MHz
Bus Reads Underway	59673279088	Bus base frequency	133.25 MHz
Reads From Processor	96043696	Bus ratio	22.51
Bus Read request latency	621.31(>300)	Bus frequency	533 MHz
Bus Writes Underway	18336719480	Theoretical bandwidth	4264 Mb/s
Writes From Processor	50658762	Bus bandwidth	2943.40 Mb/s

Таким образом, видно что последовательно выполненные повороты двух картинок в памяти основательно нагружают шину. Проанализируем теперь работу многопоточного приложения.

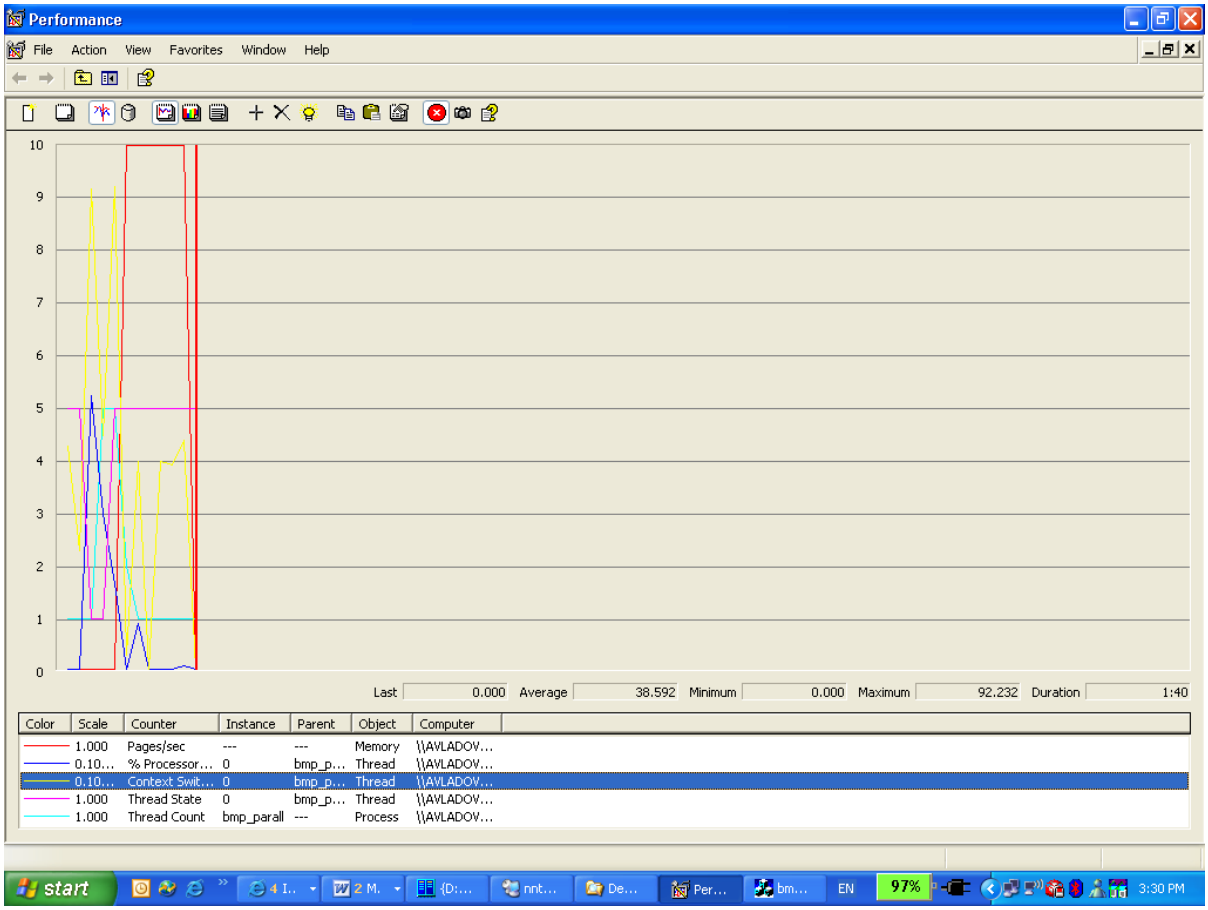


Рисунок 11 - Запуск шестипоточной программы bmp_parall.exe на Mobile

Для оценки загруженности процессоров и шины данных работа приложения проанализирована на одно- и многоядерной архитектуре с помощью инструмента Microsoft Performance Monitor в соответствии с рисунком 11. Можно видеть, что в среднем из шести работают только два потока. Максимальное количество рабочих потоков равно пяти без учета основного потока.

Таблица 6 – Результаты мониторирования шести поточной программы с помощью Performance Monitor на Mobile

Событие	Среднее значение	Максимальное значение
% Загрузка процессора *0.1	7.102	9.548
Количество потоков	2	5
Переключение потоков/сек	385.92	922.32
Количество загрузок страниц памяти/сек	46.077	96.811

Для диагностики работы программы с помощью Performance Monitor выберем счетчики загруженности процессоров, количество рабочих потоков, количества переключений между потоками и промахов страниц памяти.

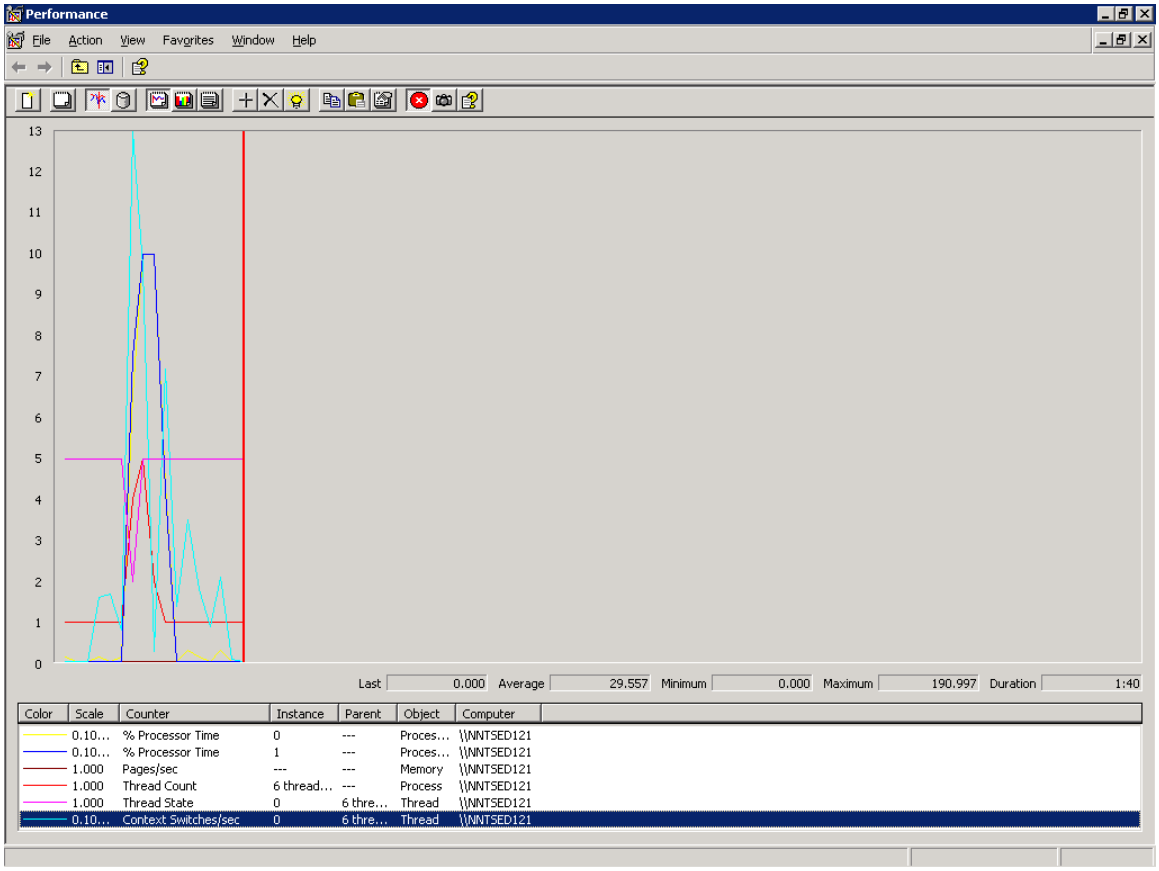


Рисунок 12 – Запуск шестипоточной версии на Xeon

Таблица 9 – Результаты мониторирования шестипоточной программы с помощью Performance Monitor на Xeon

Событие	Среднее значение	Максимальное значение
% Загрузка 1 процессора	25.358	100
% Загрузка 2 процессора	25.748	100
Количество потоков	2	5
Переключение потоков/сек	27.329	123.074
Ошибочных загрузок страниц памяти/сек	9130.891	64054.593

Возникает вопрос, почему из 6 потоков активны только 2? Проверим распределение работы между потоками с помощью Thread Profiler (рисунок 13).

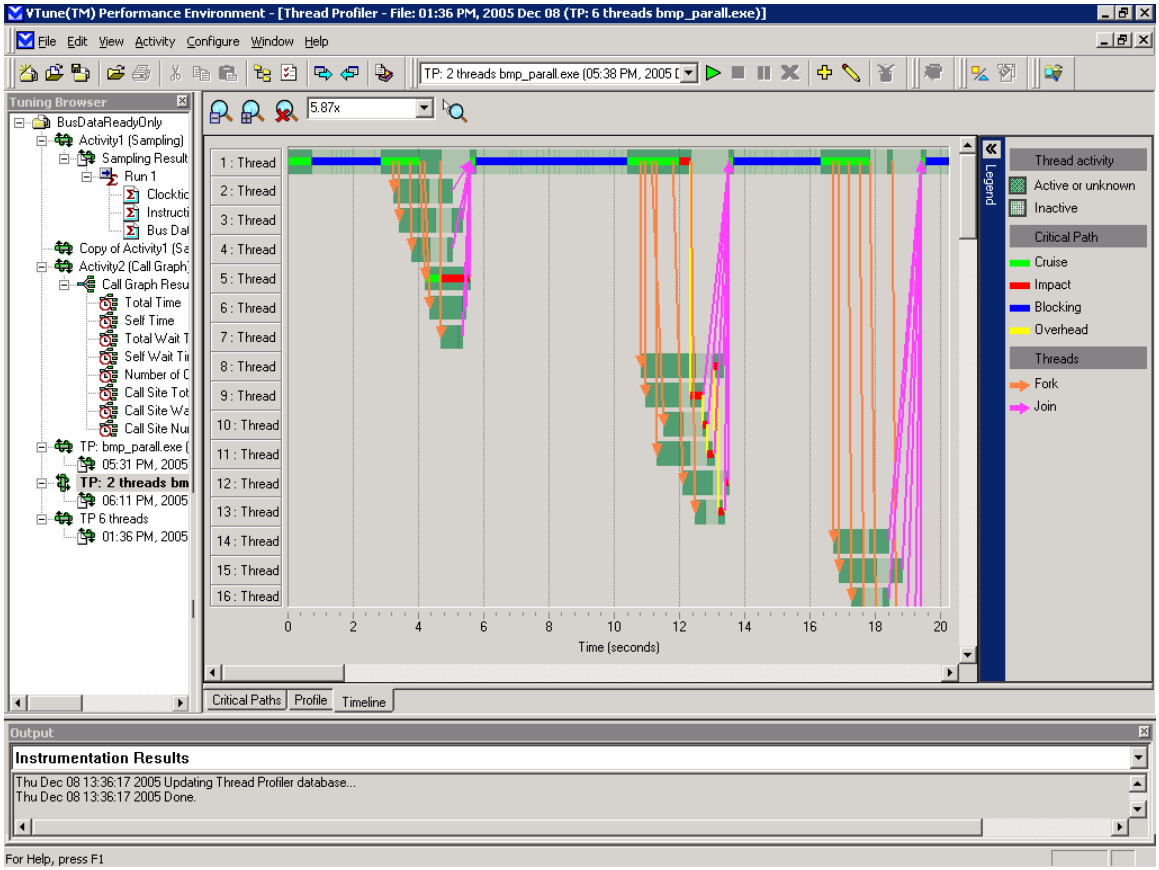


Рисунок 13 – Работа шестипоточной версии

Можно видеть, что потоки борются за ресурс и большое количество времени простаивают, а также накладные расходы велики при разделении и слиянии потоков. Сравним распределение времени работы между шестью и двумя потоками. Как видно на рисунке 14, работа распределена значительно лучше, хотя и присутствует элемент борьбы за критический ресурс.

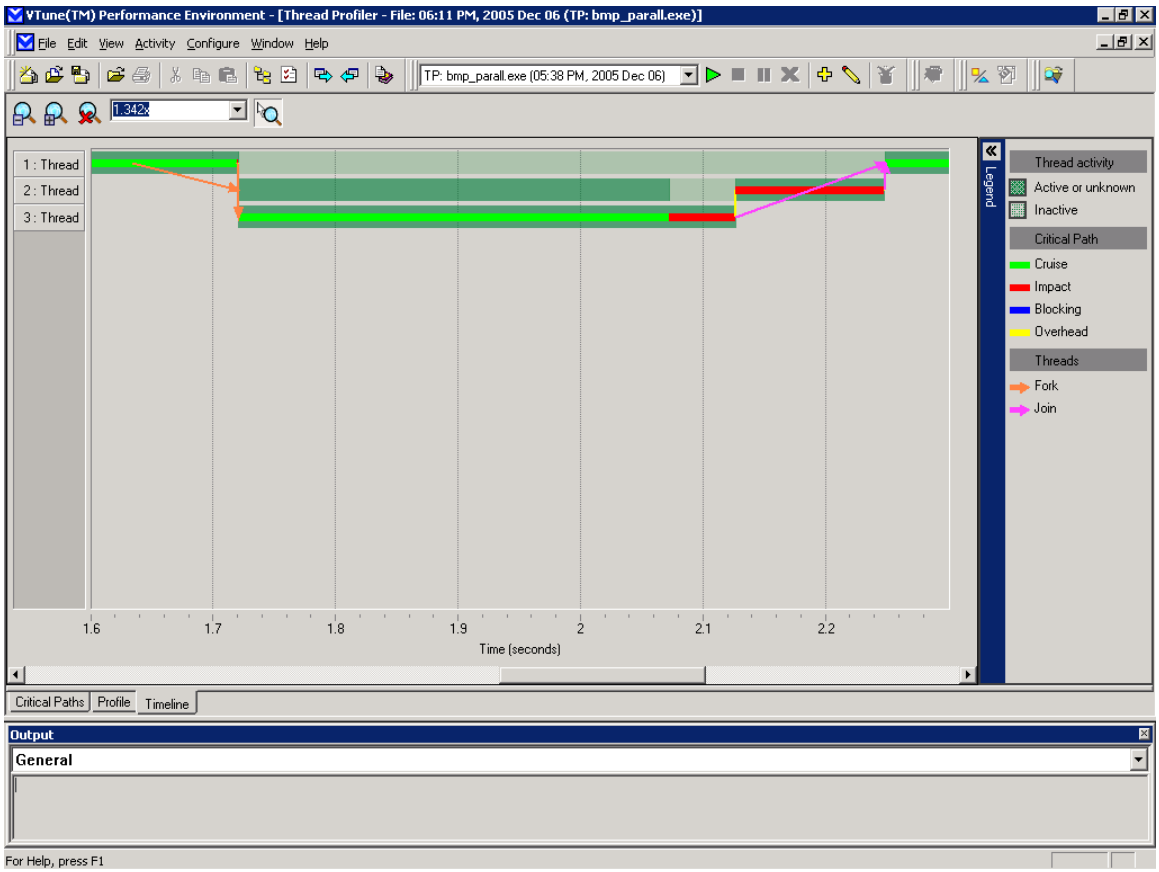


Рисунок 14 – Работа 2-х поточной версии

Таблица 10 – Результаты мониторирования двухпоточной программы на Xeon

Событие	Среднее значение	Максимальное значение
% Загрузка 1 процессора	67.71	78.125
% Загрузка 2 процессора	66.22	81.250
Переключение потоков/сек	15.335	140.026
Количество потоков	1	3
Промахов страниц памяти/сек	593.815	10332.271

Таким образом, делаем вывод о нерациональности использования большего количества потоков чем число процессоров, если 2 потока утилизируют ресурсы процессоров почти на сто процентов. По графикам загрузки рисунка 15 видно, что процессоры используются равномерно, а нагрузка на шину достаточно большая и следовательно, мы справились с задачей загрузки шины.

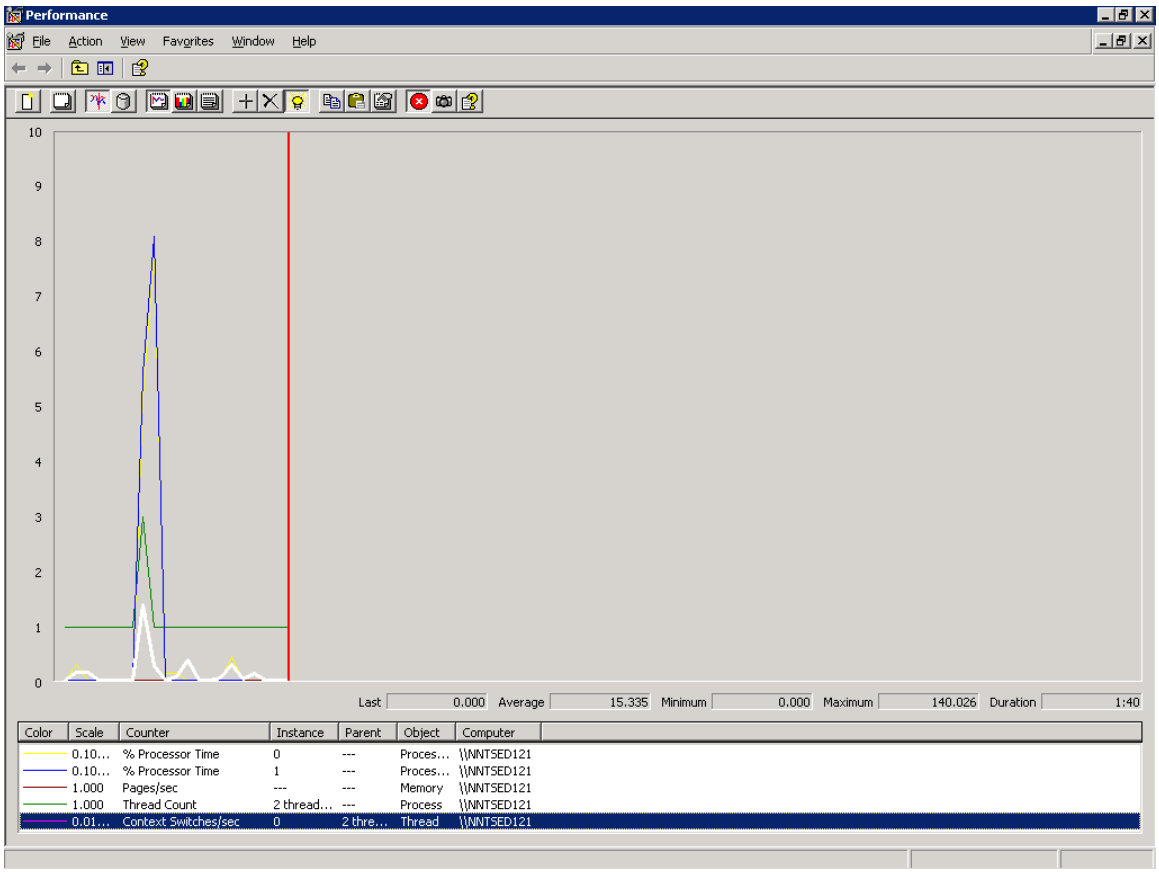


Рисунок 15 - Запуск 2-х поточной версии на Xeon

Таблица 15 – Анализ загруженности шины для 2-х поточной программы

Событие	Значение
Bus Data Ready	33890624
Clockticks	1036308000
Bus bandwidth	3139.51 MB/s
Bus ratio real	73.63 %

Для оценки загруженности шины проведем сбор событий на компьютере, оснащенном четырехпроцессорным Xeon с помощью VTune Analyzer. После запуска сбора информации получены данные, отраженные в таблицах 11-12.

Таблица 12 – Расчет загрузки шины для 4-х поточной программы

Событие	Значение
Bus Data Ready	307912380
Clockticks	8853200000
Bus bandwidth	3338.86 Mb/s
Bus ratio real	78.30 %

Таким образом, без оптимизации операций чтения-записи повысить ускорение параллельной программы не удалось.

6.10 Выводы и рекомендации

Если потоки выполняют большой объем операций чтения-записи, то количество потоков должно быть равно количеству процессоров в системе. Делаем вывод, что при масштабировании вычислительных систем следует особое внимание уделять нагрузке шины данных. Для этого требуется:

- провести расчет рабочей нагрузки на шину данных;
- разработать тестовый вариант функции программы, активно работающей с шиной;
- оценить масштабируемость программы;
- поддерживать баланс отношения количества вычислительных операций к операциям чтения-записи;
- разбивать большие блоки входных данных на более мелкие и правильно распределять их в кэш-памяти.

При разном способе доступа к данным можно добиться повышения эффективности использования шины.

6.11 Учебное задание

Оценить влияние пропускной способности шины данных на масштабируемость программы, в которой потоки поворачивают снимки экрана в памяти, тем самым перегружая шину данных.

Для установки и настройки загрузите проект в среду MS Visual Studio 6.0 или выше. Установите количество потоков равным 2 (переменная THREADS_COUNT). Скомпилируйте проект в режиме Release.

Протестируйте проект на однопроцессорной машине в последовательном и параллельном режимах и заполните таблицу 13.

Таблица 13 – Результаты тестов

№ запуска	2 параллельных потока	2 последовательных поворота
1		
2		
3		
Среднее		
Ускорение		

Оцените ускорение по формуле (1), считая что доля последовательного кода составляет 0.465.

Сделайте прогноз масштабируемости программы, подставляя в формулу (1) количество процессоров p=3, 4, 6, 8. Постройте график.

Проведите тестирование на 4-х процессорной машине, устанавливая количество потоков равным 2,3,4 (переменная THREADS_COUNT). На рисунке 16 постройте график изменения ускорения и сравните его с теоретической оценкой.

Для предварительной оценки загрузки шины данных запустите Performance Monitor и соберите информацию по следующим счетчикам: загрузка процессоров, загрузка страниц памяти, переключение контекста, количество потоков. Чтобы инструментировать программу загрузите проект в среду MS Visual Studio 6.0 или выше. Произведите настройку проекта: выберите режим De-

bug, включите генерацию символьной информации, установите опцию компиляция с thread-safe run-time библиотеками.

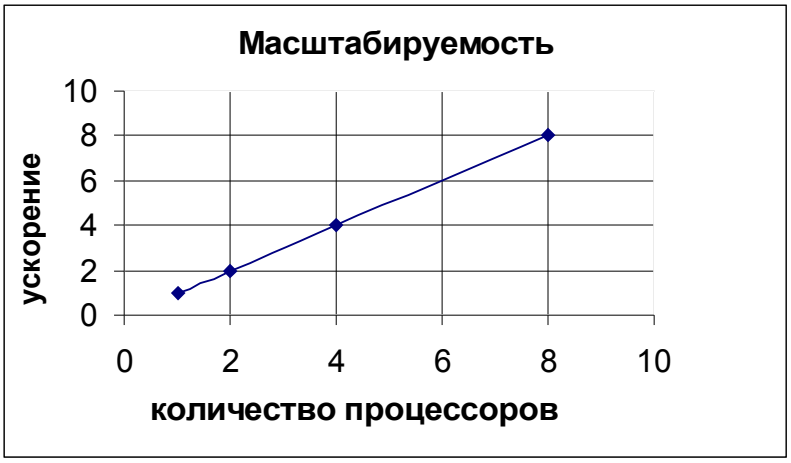


Рисунок 16 – Оценка масштабируемости программы

Запустите VTune Analyzer, Sampling. Соберите информацию по событиям шины: Bus Data Ready, Bus Reads Underway, Writes Underway, Writes и процессора: Clockticks, Instructions Retired. При анализе результатов сделайте выводы о влиянии сильной загруженности шины данных и количества потоков на масштабируемость программы.

6.12 Задание на самостоятельную работу

Сделать тестовые запуски программы на своей архитектуре и заполнить тестовые таблицы. Проанализировать узкие места программы.

Таблица 14 – Тестовые данные

№ запуска	Кол-во параллельных потоков	Кол-во пос-ных поворотов	Кол-во параллельных потоков	Кол-во пос-ных поворотов
1				
2				
3				
Среднее				
Ускорение		Ускорение		

Увеличить\уменьшить количество потоков в программе (const long THREADS_COUNT), перекомпилировать ее и оценить как влияет количество потоков на скорость работы и масштабируемость программы.

Реализовать поворот картинок с помощью библиотеки IPP и оценить скорость работы и масштабируемость программы.

С помощью VTune Analyzer промоделировать изменение тактовой частоты шины и оценить скорость работы программы.

6.13 Терминология

Параллельный блок – распараллеленный сегмент программы, который несколько потоков могут выполнять одновременно.

Последовательный блок – нераспараллеленный сегмент программы, в котором каждый оператор выполняется после другого в последовательном режиме.

Ускорение Sp параллельного алгоритма к последовательному есть отношение времени выполнения последовательного алгоритма при использовании одного процессора ко времени выполнения параллельного алгоритма при использовании p процессоров Sp = Ttotal/Tparal

Масштабируемость параллельной программы - это линейная зависимость скорости ее выполнения от производительности масштабируемой вычислительной системы.

В идеальном случае при удвоении числа процессоров программе требуется вдвое меньше времени для выполнения. Если в программе присутствуют блоки кода, которые нужно выполнить до начала выполнения других, масштабируемость таких программ ухудшается.

API – функции - это функции самой операционной системы

Снэпшот – снимок экрана

6.14 Литература, рекомендуемая для изучения раздела

1 Дж. Рихтер. Windows для профессионалов (Программирование в Win32 API для Windows NT 3.5 и Windows 95). Второе издание. М: "Русская Редакция", 1995.

2 J. Beverige, R. Wiener “Multithreading Applications in Win32: the complete guide to threads” – Addison-Wesley, 368 p.

3 Дж. Рихтер. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). Четвертое издание. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

4 Дж. Рихтер, Дж. Кларк. Программирование серверных приложений для Microsoft Windows 2000. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

5 Гергель В.П., Стронгин Р.Г. «Основы параллельных вычислений для многопроцессорных вычислительных машин»

6 Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.:БХВ-Петербург, 2002. – 400 с.:илл.

7 Методика разработки многопоточных приложений: принципы и практическая реализация Intel, RSDN Magazine #3-2004

8 Круглински, Уингоу, Шеферд «Программирование на Microsoft Visual C++ 6.0 для профессионалов»

7 Лабораторная работа №2 Влияние размера пула потоков на масштабируемость программ

Целью работы является демонстрация влияния размера пула потоков на масштабируемость программы с акцентированием на особенности организации и управления потоками.

Необходимое программное обеспечение: операционная система Microsoft Windows 2000, среда разработки Framework 1.1, программа, реализующая пул потоков

7.1 Причины использования пула потоков

Проблемы управления, создания и уничтожения потоков, рассчитанные на определенные сценарии, решаются разной реализацией пулов потоков. В Windows 2000 имеются функции для операций с пулами потоков, которые упрощают создание, уничтожение и контроль за потоками. Встроенные в них механизмы предлагают общие решения и не годятся на все случаи жизни, но зачастую их вполне достаточно, и они позволяют экономить массу времени при разработке многопоточного приложения. Рассмотрим несколько сценариев, когда оправданно использовать пул потоков.

Причина - асинхронный вызов функций. Допустим, у Вас есть серверный процесс с основным потоком, который ждет клиентский запрос. Получив его, он порождает отдельный поток для обработки этого запроса. Тем самым основной поток освобождается для приема следующего клиентского запроса. Такой сценарий типичен в клиент-серверных приложениях и его можно реализовать с использованием функций пула потоков.

Причина - вызов функций через определенные интервалы времени. Иногда какие-то операции приходится выполнять через определенные промежутки времени. В Windows имеется объект ядра «ожидаемый таймер", который позволяет легко получать уведомления по истечении заданного времени. Конечно, можно создать единственный ожидаемый таймер и каждый раз перенастраивать его на другое время ожидания. Однако такой код весьма непрост. Эту работу также можно поручить функциям пула потоков.

Причина - вызов функций при освобождении отдельных объектов ядра. Microsoft обнаружила, что во многих приложениях потоки порождаются только для того, чтобы ждать на тех или иных объектах ядра. Как только объект освобождается, поток посылает уведомление и снова переходит к ожиданию того же объекта. Создание нескольких потоков, ждущих один объект, влечет за собой возрастание накладных расходов, хотя издержки от создания потоков существенно меньше, чем от создания процессов. У каждого потока свой стек, не говоря уж об огромном количестве команд, выполняемых процессором при создании и уничтожении потока Поэтому надо стараться сводить любые издержки к минимуму. Поэтому если необходимо при освобождении какого-либо объекта ядра, поставить следующий рабочий элемент на обработку, используйте пул потоков.

Причина - вызов функций по завершении запросов на асинхронный ввод-вывод. Последний сценарий самый распространенный. Серверное приложение

выдает запросы на асинхронный ввод-вывод, и нужен пул потоков, готовых к их обработке. Это как раз тот случай, на который и были изначально рассчитаны порты завершения ввода-вывода. При управлении собственным пулом, создается порт завершения ввода-вывода и пул потоков, ждущих на этом порте. Описатели устройств ввода-вывода связаны с портом. По мере завершения асинхронных запросов на ввод-вывод, драйверы устройств помещают «рабочие элементы» в очередь порта завершения. Это прекрасная архитектура, позволяющая небольшому количеству потоков эффективно обрабатывать несколько рабочих элементов, и очень хорошо, что она заложена в функции пуля потоков. Для использования преимуществ данной архитектуры надо лишь открыть требуемое устройство и сопоставить его с компонентом поддержки других операций (не связанных с вводом-выводом), учитывая, что все потоки в этом компоненте ждут на порте завершения.

7.2 Математическое описание системы

Рассмотрим возможность математического моделирования нашей системы. Чтобы оценить вероятностные параметры функционирования системы, такие как вероятность отклонения запроса вследствие переполнения очереди, средний размер очереди, среднее количество занятых каналов и т.п., проведем описание функционирования пула потоков на основе теории моделирования непрерывно стохастических моделей. К стохастическим моделям относятся системы массового обслуживания (англ. queuing system), которые называют Q- схемами. Под СМО понимают динамическую систему, предназначенную для эффективного обслуживания случайного потока заявок при ограниченных ресурсах системы. В любом элементарном акте обслуживания можно выделить две основные составляющие: ожидание обслуживания заявкой и собственно обслуживание заявки. На каждый канал обслуживания поступают потоки событий. Потоком событий (ПС) называется последовательность событий, происходящих одно за другим в какие-то случайные моменты времени. Процесс функционирования пула можно представить как процесс изменения состояний его каналов во времени. Переход в новое состояние означает изменение кол-ва заявок, которые в нём находятся. Q-схема пула образуется композицией нескольких каналов. Если каналы соединены параллельно, то имеет место многоканальное обслуживание (многоканальная Q-схема). В общем случае модель в соответствии с рисунком 17 будет иметь следующую схему.

Рисунок 17 – Модель пула потоков

Основными параметрами являются следующие характеристики: входной поток запросов (интенсивность потока λ), характеристики обработки (интенсивность потока μ), количество каналов обработки (размер пула) (N), размер очереди (L).

Считаем, что потоки, переводящие систему из состояния в состояние простейшие, число состояний конечно, время течет непрерывно. Рассмотрим процесс функционирования системы. Система может находиться в одном из N+L+1 состояний. Состояние S_0 соответствует свободному пулу. Состояния S_1 – S_N соот-

ветственно – занятости от одного до N слотов пула. Состояния S_{N+1} – S_{N+L} соответственно занятости всего пула и наличию очереди длины от 1 до L. Система переходов между состояниями показана на рисунке 18.

Рисунок 18 – Состояния системы

На основе информации о состоянии системы составляем систему уравнений Колмогорова, описывающих вероятности нахождения в каждом из состояний.

$$\begin{cases} \lambda p_0 = \mu p_1 \\ (\lambda + \mu) p_1 = \lambda p_0 + 2\mu p_2 \\ \dots \\ (\lambda + N\mu) p_N = \lambda p_{N-1} + N\mu p_{N+1} \\ (\lambda + N\mu) p_{N+1} = \lambda p_N + N\mu p_{N+2} \\ \dots \\ (\lambda + N\mu) p_{N+L-1} = \lambda p_{N+L-2} + N\mu p_{N+L} \\ p_0 + \dots + p_{N+L} = 1 \end{cases} \quad (4)$$

После решения этой системы можно определить вероятность нахождения в любом из состояний, опираясь на информацию о работе системы, такую как интенсивности потоков (входного и обрабатывающего), размеры пула и очереди.

$$\rho = \lambda / \mu$$

$$p_1 = \rho^1 p_0$$

$$p_2 = \frac{1}{2}\rho^2 p_0$$

...

$$p_N = \frac{1}{N!}\rho^N p_0 \qquad p_0 = \left(\sum_{i=0}^{N} \frac{1}{i!}\rho^i + \frac{1}{N!}\sum_{i=1}^{L} \frac{1}{N^i}\rho^i \right)^{-1} \quad (5)$$

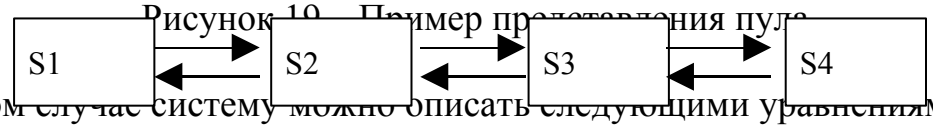
$$p_{N+1} = \frac{1}{NN!}\rho^{N+1} p_0$$

...

$$p_{N+L} = \frac{1}{N^L N!}\rho^{N+L} p_0$$

Результатом вычислений стала формула, позволяющая оценить вероятностные характеристики работы пула по имеющимся априорным параметрам. Рассмотрим пример пула с четырьмя состояниями (S1, S2, S3, S4):

Рисунок 19 – Пример представления пула



В этом случае систему можно описать следующими уравнениями Колмогорова:

$$\begin{aligned}
\frac{dp1(t)}{dt} &= \lambda 21 p2(t) - \lambda 12 p1(t) \\
\frac{dp2(t)}{dt} &= \lambda 12 p1(t) + \lambda 32 p3(t) - (\lambda 21 + \lambda 23) p2(t) \\
\frac{dp3(t)}{dt} &= \lambda 23 p2(t) + \lambda 43 p4(t) - (\lambda 32 + \lambda 34) p3(t) \\
\frac{dp4(t)}{dt} &= \lambda 34 p3(t) - \lambda 43 p4(t)
\end{aligned} \qquad (6)$$

Численные значения λij можно определить опытным путем, но для удовлетворительного моделирования требуются многочисленные тестовые испытания, что представляет собой предмет отдельного исследования, выходящий за рамки лабораторной работы.

7.3 Реализация приложения[2]

Рассмотрим основные средства, предоставляемые операционной системой MS Windows для реализации пула потоков. К наиболее низкоуровневым механизмам, предоставляемым MS Windows, можно отнести асинхронный вызов процедуры QueueUserACP(pfnAPC, hThread, dwData), позволяющий организовывать очередь из заданий на выполнение в указанном потоке, и порт завершения ввода-вывода, являющийся одним из прототипов пула, ориентированным на операции асинхронного ввода-вывода.

Начиная с MS Windows 2000, предоставляются более высокоуровневые механизмы для организации пула потоков:

- пул потоков процесса – позволяет использовать пул потоков, предоставляемый операционной системой (для каждого процесса может быть только один пул потоков операционной системы);

- работа с очередями таймеров – позволяет организовать очередь таймеров, каждый из которых вызовет срабатывание определенной процедуры заданное количество раз с заданным интервалом запуска;

- асинхронный вызов при переходе объекта в сигнальное состояние – функция RegisterWaitForSingleObject позволяет вызвать указанную процедуру после срабатывания указанного объекта, не блокируя при этом вызывающий поток.

Для работы с платформой .NET используем язык C#. При использовании среды выполнения .NET Framework существует возможность получить более высокоуровневый интерфейс к функциям работы с пулом потоков. Эти возможности дает класс System.Threading.ThreadPool, статические методы которого позволяют осуществлять работу с пулом потоков процесса, выделяемого операционной системой. Далее приведен список методов этого класса.

BindHandle – связывает хэндл операционной системы с пулом потоков

GetAvailableThreads – возвращает разницу между максимальным числом потоков и количеством потоков активных в данный момент

GetMaxThreads – возвращает максимальное количество потоков

GetMinThreads – возвращает количество потоков, которые пул поддерживает в состоянии ожидания

[2] Пример подготовлен совместно с Ковальчуком С. В.

QueueUserWorkItem – добавляет задание для пула в очередь

RegisterWaitForSingleObject – регистрирует делегата, который будет активирован при переходе указанного объекта в сигнальное состояние

SetMinThreads – устанавливает количество потоков, которые пул поддерживает в состоянии ожидания

UnsafeQueueUserWorkItem – добавляет задание для пула в очередь (небезопасный с точки зрения синхронизации вариант)

UnsafeRegisterWaitForSingleObject – регистрирует делегата, который будет активирован при переходе указанного объекта в сигнальное состояние (небезопасный с точки зрения синхронизации вариант)

Таким образом, возможности по организации пула потоков средствами MS Windows достаточно широки, но и они не лишены недостатков. К ним можно отнести следующие особенности работы системного пула потоков:

- отсутствие прямого контроля над потоками, находящимися в пуле;
- отсутствие контроля над заданиями, находящимися в очереди;
- реализация очереди без приоритетов.

7.4 Анализ предполагаемых проблем пула потоков

Рассмотрим возможные проблемы, возникающие при работе пула потоков. Ограниченность очереди к пулу приводит к тому, что при заполнении очереди запрос отклоняется. Это может привести к потере запроса или к некорректной его обработке. Ограничение очереди может быть как программно заданным параметром, так и следствием ограниченности памяти компьютера. Решением этой проблемы является организация динамической очереди, а так же изменение политики обработки непринятых запросов.

Ограниченность пула может привести к неограниченному росту очереди, что в свою очередь приведет к предыдущей проблеме. В то же время увеличение размера пула кроме положительного эффекта может дать и отрицательный – простой незанятых ячеек пула, излишняя нагрузка на процесс менеджер. Основным способом решения этой проблемы является корректный подбор размера пула, что чаще всего осуществляется экспериментальным путем. Повышенная нагрузка на поток менеджер приводит к замедлению работы всей системы, так как этот поток осуществляет обслуживание всего пула. Снизить влияние этого фактора можно распределением части нагрузки потока менеджера между рабочими потоками.

7.5 Описание алгоритма

Задания, получаемые системой для обработки, представляют собой некий диапазон целых чисел. Задание передается в качестве параметра в рабочую процедуру (WorkProc), которая в цикле, проходящем по всем числам этого диапазона, выполняет некоторую работу. В качестве подобной работы, была использована простейшая математическая операция, вычисления синуса, выполняемая достаточно большое число раз. Таким методом осуществляется загрузка процессора и эмуляция активной вычислительной работы.

Два варианта исходного текста программы соответствуют программе, выполняющей все задания последовательно и программе выполняющей операции с использованием пула потоков. Однако рабочая процедура в обоих случаях ис-

пользуется одна и та же. В первом случае осуществляется ее последовательный вызов в каждой итерации цикла. При использовании этого варианта все задания обрабатываются в основном потоке программы. Во втором случае в этом цикле осуществляется постановка задач в очередь к пулу. В случае работы с пулом программа ожидает завершения обработки всех заданий, анализируя количество доступных потоков в пуле.

И в первом, и во втором вариантах замеряется время работы программы, получая разницу системного времени в начале и в конце выполнения программы. Используя полученные результаты можно оценить ускорение и масштабируемость программы.

7.6 Практическая оценка ускорения и масштабируемости приложения

Проведем тесты на машинах с разным количеством процессоров, изменяя размер пула и подсчитывая время последовательного и параллельного режима работы программы, чтобы получить оценку ускорения и масштабируемости. Видим, что для однопроцессорной машины наиболее оптимален 2-х и 4-х канальный пул. Проведем тест на двупроцессорной машине.

Таблица 15 – Тест на однопроцессорной машине

Размер пула	Параллельный запуск, с	Послед запуск, с	Ускорение
2	18,03	20,66	1,15
4	35,46	41,31	1,16
6	53,39	56,56	1,06
8	114,86	71,24	0,62
10	135,51	88,72	0,65

Таблица 16 – Тест на двупроцессорной машине

Размер пула	Параллельный запуск, с	Послед запуск, с	Ускорение
2	8.9	15,68	1,76
4	17.06	31,37	1,84
6	25.56	47,04	1,84
8	33.02	62,75	1,90
10	46.7	68.46	1.47

Таблица 17 – Тест на четырехпроцессорной машине

Размер пула	Параллельный запуск, с	Послед запуск, с	Ускорение
2	9,11	16,01	1,76
4	11,11	32,09	2,89
6	16,02	48,09	3,00
8	16,18	64,15	3,96
10	24,6	80,14	3,26

По результатам замера на рисунке 13 построена диаграмма, на основе которой можно сделать некоторые выводы относительно работы пула потоков. Ускорение программы зависит от размера пула - если размер пула равен удвоенному числу процессоров системы, то режим работы программы оптимален и ускорение достигает расчетной величины.

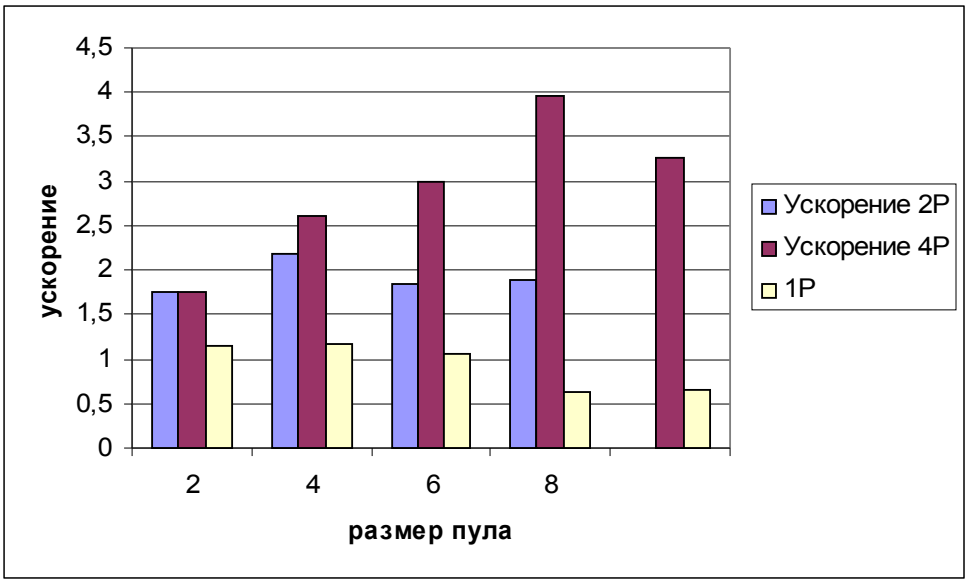


Рисунок 20 – Диаграмма результатов замеров

По графику видно, что на двух процессорах лучшее ускорение демонстрирует 4-х слотовый пул. А на 4-х процессорах – 8-ми слотовый пул. В псевдопараллельном режиме на одном процессоре наилушее ускорение достигается на 2-х слотовом пуле. Поэтому можно сделать вывод, что для оптимальной работы приложения количество слотов пула должно быть равно количеству процессоров, умноженному на два.

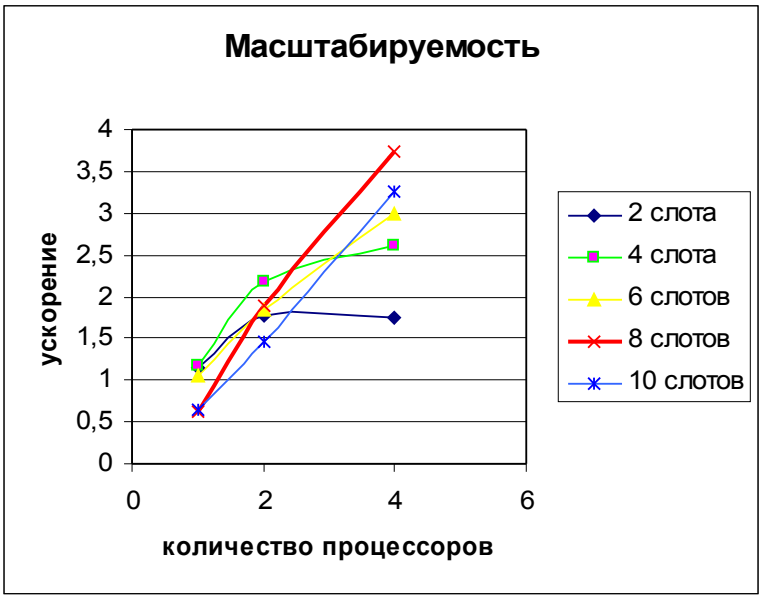


Рисунок 21 – Масштабируемость приложения

Можно заметить, что при увеличении количества процессоров и пропорциональном увеличении количества слотов пула ускорение изменяется линейно. Это связано со спецификой моделируемой задачи. Рабочий поток стремится получить все процессорное время, никогда не простаивая в ожидании каких-либо событий. В результате при выполнении в псевдопараллельном режиме нескольких потоков они выполняются медленнее и их суммарное время остается примерно тем же. Очевидно, что при дальнейшем росте числа потоков расчетное ускорение будет стремиться к 0,5. С учетом погрешности данные замеров подтверждают теоретические расчеты. То есть оптимальный прирост получается при четном числе потоков в пуле.

7.7 Выводы и рекомендации

Таким образом, можно говорить о том, что при использовании пула потоков следует уделять существенное внимание подбору размера пула, поскольку некорректный размер может не дать ожидаемого прироста времени работы. Кроме того, следует отметить, что при наличии в работе вычислительного потока периодов простоя рост размера пула приводит к более эффективной обработке запросов и росту общей производительности. Это связано со степенью загруженности процессора при работе потока обработки запроса.

Использование пула потоков оправдано в системах массового обслуживания. Примером такой системы может служить система приема платежей, обрабатывающая запросы на платежи при помощи пула потоков. В качестве примера подобной системы можно привести платежную систему X-plat (http://x-plat.ru), сервер которой работает по схожему принципу.

7.8 Учебное задание

Оценить влияние размера пула потоков на масштабируемость программ

Потоки предлагаемой программы организованы в пул. Каждый поток выполняет определенную работу над диапазоном чисел. Для установки и настройки при запуске проекта введите требуемый диапазон (от 10 до 500).

Таблица 18 – Результаты тестов

Размер пула	Параллельный запуск, с	Последовательный запуск, с	Ускорение
2			
4			
6			
8			
10			

Это число будет поделено на количество диапазонов (по умолчанию равное 10) и получена размерность пула.

Протестируйте проект в последовательном и параллельном режимах и заполните таблицу 18.

Оцените ускорение по формуле (1), измерив долю последовательного кода.

Повторите тестирование с на иной архитектуре с пропорциональным изменением размера пула, используя правило – количество каналов обработки пула равно количеству процессоров системы, умноженному на два.

Для предварительной оценки запустите Performance Monitor и соберите информацию по следующим счетчикам: загрузка процессоров, переключение контекста, количество потоков.

При инструментировании программы загрузите проект в среду MS Visual Studio .NET. Произведите настройку проекта: выберите режим Debug, включите генерацию символьной информации, установите опцию компиляция с thread-safe run-time библиотеками. Добавьте в Properties - C++ - Linker – Command Line опцию /fixed:no. Запустите Intel VTune Analyzer. Выберите режим CallGraph и произведите сбор информации о дереве вызова функций.

Для оценки результатов постройте график зависимости ускорения от размера пула на разных количествах процессоров. Сделайте вывод о влиянии размера пула на масштабируемость программы.

7.9 Задания на самостоятельную работу

1 С помощью пула потоков реализовать вызов функций через определенные интервалы времени. Оценить масштабируемость программы.

2 С помощью пула потоков реализовать вызов функций при освобождении отдельных объектов ядра. Оценить масштабируемость программы.

3 В рамках пула создать реализацию вызовов функций по завершении запросов на асинхронный ввод-вывод. Оценить масштабируемость программы.

7.10 Литература, рекомендуемая для изучения раздела

1 Э. Таненбаум, М. Ван Стеен «Распределенные системы. Принципы и парадигмы»

2 Microsoft Software Developers Network [Электронный ресурс]. - Режим доступа: http://msdn.microsoft.com/

3 Дж. Рихтер. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). Четвертое издание. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

4 А.Ширшов Эффективная многопоточность. Организация пула потоков. RSDN Magazine №2, 2003.

5 Э. Брэдфорд, Л. Можэ «Кроссплатформенные приложения для Linux иWindows»

6 А. Вильямс «Системное программирование в Windows 2000»

7 Д. Соломон, М. Руссинович. Внутреннее устройство Microsoft Windows 2000. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

Заключение

До момента массового производства двух- и многоядерных процессоров разработка параллельных программ велась ограниченным кругом лиц преимущественно для научных задач. Но процесс переноса программ на многоядерную систему сопряжен с рядом трудностей. Одной из них может быть неудовлетворительная разница между теоретически оцененным и практически полученным ускорением и масштабируемостью программы.

На примере двух задач – поворота изображений и реализации пула потоков рассматривается процесс разработки и анализа многопоточной программы, оценивается ускорение и масштабируемость, даются рекомендации по улучшению кода. С помощью библиотеки Pthreads и инструментов Intel VTune Analyzer, Intel Thread Profiler, Microsoft Performance Monitor поэтапно преодолеваются проблемы практического создания эффективных многопоточных программ.

Приведенные сведения позволят учесть аппаратные факторы, вносимые особенностями архитектуры и количественными характеристиками устройств; программные факторы, представляющие собой неэффективную реализацию алгоритма, как например, неудачный способ организации, размещения и обмена данными, низкую эффективность распараллеливания и др.

Список использованных источников

1 Л. Черняк Ядра и потоки современных микропроцессоров / Открытые системы, № 03, 2006.

2 Двухядерные технологии от AMD [Электронный ресурс] / AMD - Режим доступа: http://vpr.ocs.ru/dual-core.html.

3 Процессоры Intel и AMD [Электронный ресурс] - Режим доступа: http://www.compress.ru/Archive/CP/2005/12/3/.

4 Микроархитектура Intel® Core™ послужит основой для новых экономичных высокопроизводительных компьютеров. Форум Intel для разработчиков, Сан-Франциско, 2006 г. [Электронный ресурс] - Режим доступа: http://www.citcity.ru/11952/.

5 Двухъядерные процессоры Intel: выбираем лучший. [Электронный ресурс] / Обзоры и тесты. 2005. - №12. - Режим доступа: http://www.ferra.ru/online/processors

6 А. Рыбаков, С. Золотарев Программное обеспечение многоядерных систем / Открытые системы, № 02, 2006 [Электронный ресурс] - Режим доступа: http://www.osp.ru/text/302/1156508/

7 А. Борзенко Многоядерные процессоры [Электронный ресурс] / - 2005. - №3. - Режим доступа: http://www.bytemag.ru/?ID=603854

8 Процессоры Intel и AMD [Электронный ресурс] // КомпьютерПресс -2005 . №12. - Режим доступа: http://www.compress.ru/Archive/CP/2005/12/3/.

9 Корнеев В. В. Вычислительные системы – М.:Гелиос АРВ, 2004. – 512 с.

10 Богачев К. Ю. Основы параллельного программирования – М.:БИНОМ. Лаборатория знаний, 2003. -342 с., илл.

11 Архитектуры и топологии многопроцессорных вычислительных систем. Курс лекций. Учебное пособие / А. В. Богданов, В. В. Корхов, В. В. Мареев, Е. Н. Станкова / - М.ИНТУИТ.РУ «Интернет-Университет Информационных Технологий», 2004. -176 с.