Министерство образования и науки РФ

Нижегородский государственный университет им. Н.И. Лобачевского

К.В. Корняков, И.Б. Мееров, А.А. Сиднев, А.В. Сысоев, А.В. Шишков

ИНСТРУМЕНТЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ

Учебное пособие

Нижний Новгород
Издательство Нижегородского госуниверситета
2010

ББК 32.973.26−018.2:22
УДК681-3-06
И 38

Корняков К.В., Мееров И.Б., Сиднев А.А., Сысоев А.В., Шишков А.В.
И 38 **Инструменты параллельного программирования в системах с общей памятью. –** Учебное пособие / Под ред. проф. В.П. Гергеля. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2010. – 201 с.

ISBN 978-5-91326-138-0

В пособии описываются методы и инструментальные средства для разработки, отладки и профилировки параллельных программ, ориентированных на работу в системах с общей памятью. Рассматриваются программные пакеты Intel Thread Checker, Intel Thread Profiler, Intel Threading Building Blocks. Демонстрируется весь цикл работ, начиная от создания последовательной реализации в качестве базы для сравнения, подготовки параллельной версии, ее отладки, профилировки и оптимизации. Изучение проводится на модельных задачах, не требующих наличия специфических знаний из конкретных предметных областей. Подразумевается знакомство читателя с основами программирования (базовый язык – C/C++), некоторые задачи требуют сведения из высшей математики (задача Дирихле).

Учебное пособие разработано в лаборатории «Информационные технологии» (ITLab) факультета ВМК ННГУ на основе материалов, подготовленных в рамках Приоритетного национального проекта «Образование».

Для преподавателей и научных сотрудников, а также аспирантов и студентов высших учебных заведений.

ISBN 978-5-91326-138-0 ББК 32.973.26−018.2:22

Издано при поддержке корпорации «Интел»

Содержание

Предисловие

Не секрет, что вычислительная техника и технологии ее использования развиваются с большой скоростью на протяжении последних 50 лет. Одна из очевидных тенденций последнего десятилетия – смещение акцентов в сторону параллелизма. Мир вычислительной техники становится параллельным!

В настоящем пособии мы ориентируемся на системы с общей памятью, за короткое время проделавшие путь от исследовательских образцов до стандартных настольных систем. Как это нередко бывает, аппаратная часть несколько опережает в развитии программную, в том числе и системного уровня. Так, на сегодня программистское сообщество может похвастаться десятками моделей, технологий, языков и библиотек для параллельного программирования, но мы все еще очень далеки от того, чтобы сделать его доступным не только избранным, но, напротив, всей отрасли. В частности, достаточно важно «вооружить» подобными технологиями и средствами прикладных программистов – тех, кто решает ресурсоемкие задачи в конкретных проблемных областях – физике, химии, биологии, экономике, медицине. Именно с учетом этого написано настоящее пособие.

Пособие предполагает наличие у читателя базовых знаний и навыков структурного, модульного и (в последнем разделе) объектно-ориентированного программирования. В качестве базового языка рассматривается C/C++. Разумеется, многие идеи могут быть с успехом перенесены на другой язык, поддерживающий многопоточное программирование, в частности Fortran.

Пособие построено следующим образом: в первой части изучается отладка параллельных программ, рассматриваются основные виды ошибок, приводится описание отладчика Intel Thread Checker, изучается методика его использования в ходе выполнения лабораторной работы. В процессе решения модельных задач демонстрируются основные ошибки, методы их обнаружения и устранения. Во второй части изучаются производительность и оптимизация параллельных программ, рассматривается профилировщик Intel Thread Profiler, приводится методика профилировки и оптимизации; в лабораторных работах производится демонстрация возможностей инструмента. В третьей части изучается разработка параллельных программ при помощи библиотеки Intel Theading Building Blocks, приводится описание возможностей библиотеки и методики создания и оптимизации параллельных программ.

По многим разделам приводятся задания для самостоятельной проработки и контрольные вопросы.

Макет книги (цветная версия), примеры программ, необходимые для выполнения лабораторных работ, а также материалы разделов в виде отдельных документов доступны для скачивания по адресу: http://www.software.unn.ru/ccam/multicore/programm.html.

Часть I. Отладка параллельной программы

1. Intel Thread Checker. Краткое описание

Ни для кого не секрет – наличие инструментов делает жизнь проще. Тезис этот находит подтверждение как в повседневной жизни (чистить одежду удобнее и эффективнее щеткой, чем руками, заворачивать гайки гораздо проще ключом, чем пальцами), так и в профессиональной деятельности (кто сейчас возьмется строить, ладно, даже не дом, пусть всего лишь баню, при помощи одного лишь топора?!). Естественно, программисты – не исключение. Значение инструментов, облегчающих путь от анализа постановки задачи до получения решения, готового к внедрению, трудно переоценить. В данном кратком описании представлен инструмент отладки параллельных программ, разработанный корпорацией Intel и носящий название Intel® Thread Checker.

В п. 1.1 приводится назначение рассматриваемого инструмента, характеризуются области его возможного применения. В п. 1.2 дается краткая характеристика принципов работы Intel® Thread Checker. В пп. 1.3 – 1.5 приводится информация, необходимая для подготовки пользовательского проекта и инструмента для анализа. П. 1.6 посвящен вопросам сбора и анализа данных, полученных в результате работы Intel® Thread Checker. В п. 1.7 возможности инструмента рассмотрены на простом примере, входящем в поставку Intel® Thread Checker.

1.1. Назначение Intel Thread Checker

Процесс отладки в общем случае можно разбить на следующие шаги:

- определение факта наличия ошибки;
- поиск (локализация) ошибки;
- выяснение причин ошибки;
- определение способа устранения ошибки;
- устранение ошибки.

Кажется, что на первом шаге никакой инструмент не требуется. Запускаем программу и либо на некоторых исходных данных получаем неверные результаты, либо обнаруживаем, что некоторая последовательность действий по использованию программы ведет к ее «падению» или «зависанию». Однако для параллельной программы даже этот очевидный шаг может иметь существенную сложность. На практике нередко встречаются ситуации, когда неработоспособность параллельной программы проявляется один раз на сотню и более запусков. Очевидно, в этом случае инструментальная поддержка лишней не будет.

Назначение Intel® Thread Checker (ITC) – поиск мест с возможным недетерминированным поведением многопоточной программы, написанной как на основе библиотеки потоков (Windows или POSIX threads), так и с использованием технологии OpenMP. Соответственно ITC может быть использован под операционными системами как семейства Windows, так и семейства Linux. Принципы поиска ошибок рассмотрены ниже, здесь же укажем, что ITC неплохо справляется с задачей обнаружения факта ошибки в программе, даже если эта ошибка в текущем варианте исполнения программы и не проявила себя.

Второй шаг – поиск ошибки – заключается в как можно более точной ее локализации; в идеале должна быть найдена переменная с неверным значением и/или строка кода, ведущая к краху программы. Типичный метод работы в этом пункте – использование режима трассировки в отладчике с наблюдением за состоянием переменных, регистров, стека вызова и т.д. «Плохая новость» – для многопоточных программ режим трассировки практически неприменим, поскольку автоматически меняет характер их выполнения, а значит, скрывает места, которые могут приводить к проблемам во время реальной работы. Кстати говоря, даже типичный способ локализации ошибки расстановкой операторов печати по тексту программы в этом случае нужно использовать с большой осторожностью – печать также вносит синхронизацию в выполнение программы.

Что же делать? Быть может, наилучшее из возможных решение реализовано в ITC. ITC не есть привычный всем отладчик с режимами трассировки, наблюдения и т.д. ITC выполняет анализ программы сам, без участия программиста, причем анализируется не только выполненный «прогон» программы, а все возможные варианты ее выполнения. В результате выясняются и показываются программисту места в программе, в которых содержатся ошибки (с той или иной долей вероятности, в большинстве случаев близкой к 100%).

Шаг третий – выяснение причин ошибки. Задача здесь – понять, почему ошибка возникла. Отсюда во многих случаях автоматически вытекает способ ее устранения (задача шага четвертого). Можно, конечно, выяснить условия, ведущие к проявлению ошибки (некое сочетание значений переменных, например), и просто вставить в код заплатку именно для этого случая. Данный вариант мы здесь не рассматриваем. На этом шаге ITC помогает тем, что каждое найденное им проблемное место сопровождает комментарием, содержащим тип ошибки: гонка данных, несинхронизированный доступ к переменной, тупик и т.д.

В результате мы получаем место потенциальной ошибки, переменную, с которой связана проблема, и описание ошибки. Остается лишь освоить типовые способы борьбы с типовыми ошибками – и значительная их часть будет находиться и исправляться без грандиозных усилий.

Единственное, в чем ITC совсем не может помочь – это шаг пятый. Устранять найденную ошибку все-таки придется программисту самостоятельно.

1.2. Возможности Intel Thread Checker

Согласно [2] ITC обнаруживает ошибки следующих видов: *гонки данных* (*data races*), *тупики* (*deadlocks*), *потоки в состоянии ожидания* (*stalled threads*), *потерянные сигналы* (*lost signals*), *заброшенные замки* (*abandoned locks*).

Приведем краткое описание каждого вида.

- **Гонки данных**. Возникают, когда несколько потоков работают с разделяемыми данными и конечный результат зависит от соотношения скоростей потоков. Пусть, например, один поток выполняет над общей переменной `x` операцию `x = x + 3`, а второй поток – операцию `x = x + 5`. Данные операции для каждого потока фактически разбиваются на три отдельные подоперации: считать `x` из памяти, увеличить `x`, записать `x` в память. В зависимости от взаимного порядка выполнения потоками подопераций финальное значение переменной `x` может быть больше исходного на 3, 5 или 8. Гонка данных возможна и в случае, когда один поток пишет в переменную, а остальные только читают из нее.

- **Тупики**. Взаимная блокировка потоков, ожидающих наступления некоторого события для продолжения работы. Типичный пример тупика: нулевой поток занял для использования ресурс 1 и ожидает предоставления ему ресурса 2, а первый поток занял ресурс 2 и ожидает предоставления ему ресурса 1.
- **Потоки в состоянии ожидания**. Одно из состояний потока в многозадачной операционной системе – ожидание. Поток переходит в него, когда для продолжения выполнения ему требуется наступление некоторого внешнего события. Если пребывание потока в этом состоянии продолжается слишком долго, ITC рапортует об ошибке типа stalled thread. Интервал времени, по истечении которого выдается данная диагностика, может быть задан в настройках ITC.
- **Потерянные сигналы**. Возникают, когда поток ожидает наступления некоторого события, произошедшего прежде, чем поток пришел в состояние готовности к его приему и обработке. В результате поток никогда не сможет выйти из состояния ожидания.
- **Заброшенные замки**. Возникают в ситуации, когда поток захватил некоторый ресурс (критическую секцию, мьютекс) и был снят с выполнения по той или иной причине. В результате ресурс не может быть освобожден. Если он требуется другому потоку, это приведет к бесконечному ожиданию.

1.3. Принцип сбора информации

Анализ программы, выполняемый ITC, основан на процедуре инструментации. *Инструментация* – вставка обращений к библиотеке ITC для записи действий, потенциально способных привести к ошибкам: работа с памятью, вызовы операций синхронизации и работа с потоками [2]. Может выполняться автоматически на уровне исполняемого модуля (а также dll-библиотеки) и/или по указанию программиста на уровне исходного кода. Для достоверности получаемых результатов крайне желательно, чтобы во время сборки анализируемой программы была выключена оптимизация (сборка в конфигурации **debug** необязательна).

В процессе анализа контролируются:

- доступ к памяти;
- операции синхронизации;
- операции создания потоков.

Необходимо отметить, что неисполняемые участки (невызываемые функции, ветки условных переходов и т.д.) никак не проверяются, то есть под анализ не подпадают.

1.4. Подготовка программы для анализа

Использование отладчика Intel® Thread Checker возможно в двух режимах:

- *Бинарная инструментация программы* – осуществляется автоматически в момент запуска **Активности**[1] (**Activity**) в проекте ITC. Рекомендуется в случае, если отсутствует доступ к исходным кодам или невозможна повторная сборка программы с нужными ITC настройками.

[1] Активность – термин из среды VTune, в рамках которой работает ITC. Фактически представляет собой «контейнер», содержащий, с одной стороны, настройки системы и параметры анализируемой программы, с другой – результаты проводимого анализа.

- *Компиляторная инструментация* – при сборке анализируемой программы необходимо указать ключ компилятора **/Qtcheck.** Позволяет ITC предоставить информацию о найденных ошибках с указанием имен переменных, с которыми эти ошибки связаны.

Сборка приложения для работы с ITC предполагает установку следующих опций проекта (или настроек в make-файле):

- Компиляция потокобезопасного кода: **-MT[d]**, **-MD[d]**. Данные опции автоматически устанавливаются при сборке в конфигурации **debug**. При сборке в конфигурации **release** указанные опции необходимо устанавливать вручную.
- Использование debug-опций: **-Z[i,I,7]**, **-Od**. Замечание аналогично предыдущему пункту.
- Связывание с ключом **/fixed:no**. Необходимо указывать явно.

Дополнительно необходимо отметить, что при использовании ключа **/Qtcheck** в среде разработки (IDE) требуется указать путь к библиотекам ITC. Обычно этот путь имеет вид **C:\Program Files\Intel\VTune\Analyzer\Lib\.**

1.5. Создание проекта в Intel Thread Checker

Работа в ITC выполняется в рамках **проекта**. Для его создания используется команда меню **File→New Project**. В главном окне мастера настройки проекта (см. рис. 1) необходимо выполнить всего лишь два действия. Первое – указать исполняемый файл (**Launch an application**). Второе (необязательное) – указать аргументы командной строки**.**

Рекомендуется при анализе программы, с одной стороны, использовать типичные размеры обрабатываемых данных, с другой – задавать их так, чтобы программа «убиралась» в оперативную память с учетом накладных расходов ITC, которые могут быть довольно значительными.

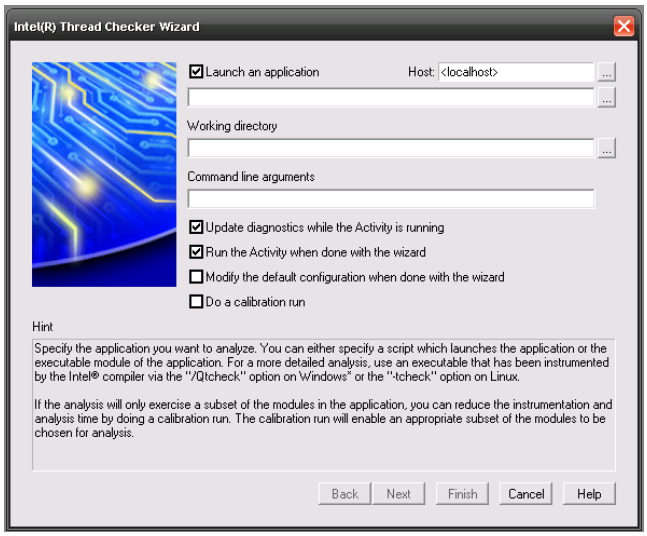

Рис. 1. Мастер настройки проекта в Intel Thread Checker (версия 3.0)

1.6. Сбор и анализ данных

После запуска в проекте ITC активности (при создании проекта это происходит автоматически) начинается инструментация исполняемого модуля, указанного для анализа, и используемых им динамических библиотек. Затем модуль запускается и начинается процесс анализа. По завершении ITC формирует окно с информацией о найденных ошибках и подозрительных местах. Возможный его вид указан на рис. 2.

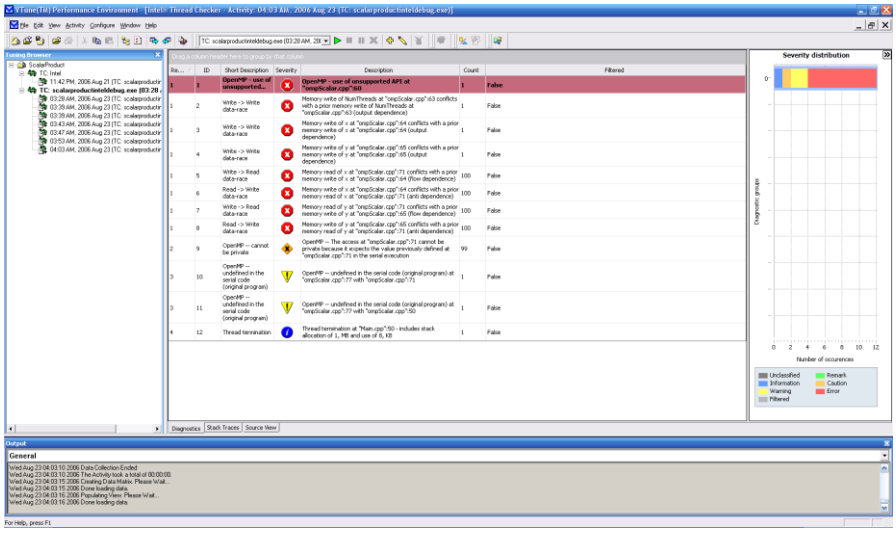

Рис. 2. Результат анализа – список диагностики

В случае повторного запуска активности необходимо использовать один из следующих вариантов: 1) выбрать пункт меню **Activity→Run**; 2) нажать **F5;** 3) нажать кнопку на панели инструментов.

По каждой диагностике, выданной ITC, в случае если сборка выполнялась с приведенными в пункте 1.4 настройками, может быть получена дополнительная информация (см. пункт 1.7), вид которой показан на рис. 3.

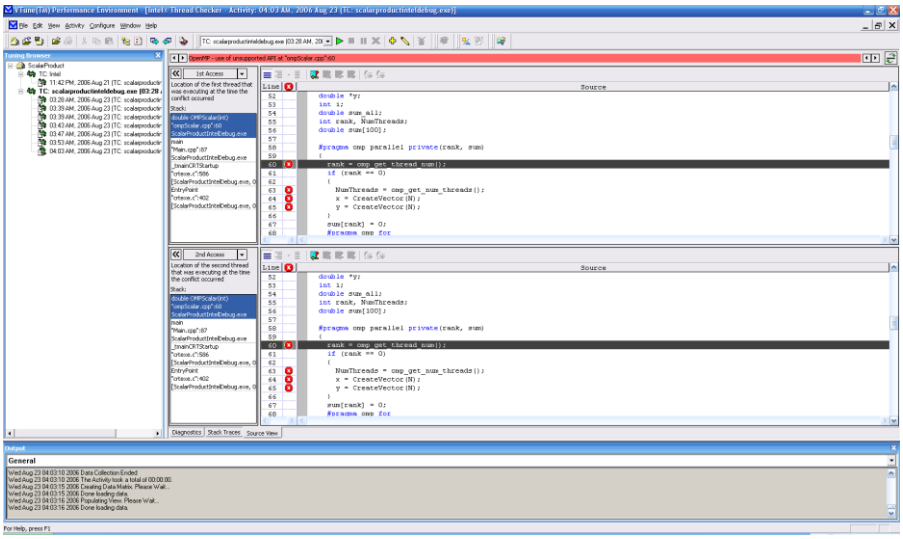

Рис. 3. Результат анализа – диагностика в исходном коде

1.7. Пример использования Intel Thread Checker

Для начального знакомства с ITC рассмотрим пример, входящий в поставку инструмента версии 3.0 и описанный в [3].

1.7.1. Описание примера

Наличие у каждого потока в многопоточном приложении доступа к общему ВАП[2] процесса позволяет потокам эффективно обмениваться данными, с одной стороны, и является основным источником ошибок, с другой. Гонки данных, они же *конфликты доступа* (*storage conflicts*), поджидают зазевавшегося программиста буквально на каждом шагу. Поиск таких ошибок методом «пристального взгляда» – задача весьма сложная.

Рассматриваемый пример создает 4 потока, каждый из которых увеличивает значение общей глобальной переменной **`globalX`** и использует критическую секцию для синхронизации доступа к ней. Однако, несмотря на наличие критической секции, в коде все-таки содержится конфликт доступа. Нам предстоит его обнаружить, выяснить причину и исправить ситуацию.

1.7.2. Изучение примера

Откройте проект **DataRaces**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ITCLabs\DataRaces**;
- дважды щелкните на файле **DataRaces.dsw** или, выбрав файл, выполните команду **Open**;
- согласитесь с предложением конвертировать проект **DataRaces.dsp** в новый формат.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **DataRaces.c**, как это показано на рис. 4. После этих действий программный код, с которым предстоит работать, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

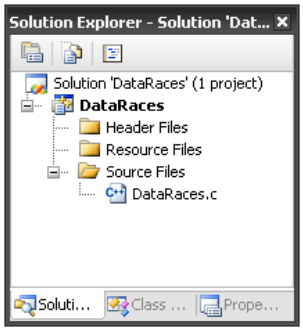

Рис. 4. Открытие файла DataRaces.c

Интерес в этом небольшом файле представляет потоковая функция **`increment`**.

[2] ВАП – виртуальное адресное пространство процесса в операционной системе.

```
int globalX = 0;

DWORD WINAPI increment (void *arg)
{
  CRITICAL_SECTION cs;

  InitializeCriticalSection (&cs);

  EnterCriticalSection (&cs);
    globalX++;
  LeaveCriticalSection (&cs);

  DeleteCriticalSection (&cs);

  return 0;
}
```

На первый взгляд код выглядит корректно. Доступ к переменной **globalX** защищен. Посмотрим, что покажет запуск программы.

Соберите и запустите пример, выполнив следующие действия:

- в меню **Build** выполните команду **Build Solution**;
- проигнорируйте предупреждение компилятора Microsoft о неизвестной опции **/Qtcheck**;
- в меню **Debug** выполните команду **Start Without Debugging**.

Убедитесь, что вывод на экран соответствует представленному на рис. 5.

Рис. 5. Результаты работы примера DataRaces (исходный вариант)

Кажется, все верно. Число потоков равно четырем. Начальное значение переменной **globalX**, как нетрудно убедиться, равно нулю. Таким образом, результат верен. Повторите запуск программы несколько раз и убедитесь, что результат стабильно равен 4.

Может быть, программа корректна? Подумаем, что нужно для того, чтобы гонки данных могли проявиться. Необходимо, чтобы имел место разный порядок выполнения потоков во времени. Посмотрим, возможно ли это в данном примере. Конечно же, нет! Вот участок функции **main**, запускающий потоки:

```
  for (i = 0; i < NTHREADS; i++)
  {
    h[i] = CreateThread (0, 0, increment, NULL, 0, NULL);
  }
```

Нетрудно понять, что потоки создаются последовательно, по мере выполнения цикла. Из-за крайней простоты, а значит, и малого времени выполнения потоковой функции скорее всего и работа потоков происходит последовательно. Конфликт доступа к переменной **globalX**, даже если он имеет место, не успевает проявиться.

Что ж, изменим немного код. Пусть каждый поток увеличивает значение переменной **globalX** не один раз, а многократно.

```
DWORD WINAPI increment (void *arg)
{
  int i;
  CRITICAL_SECTION cs;

  InitializeCriticalSection (&cs);

  for (i = 0; i < 10000; i++)
  {
    EnterCriticalSection (&cs);
    globalX++;
    LeaveCriticalSection (&cs);
  }

  DeleteCriticalSection (&cs);

  return 0;
}
```

Добавьте в код выделенные в предыдущем фрагменте строки, соберите и запустите пример. Убедитесь, что вывод на экран соответствует представленному на рис. 6.

Рис. 6. Результаты работы примера DataRraces (версия с циклом)[3]

Не правда ли, неожиданно?! Вместо 40000 на экране совсем другое число! Повторите запуск несколько раз и убедитесь, что результат работы программы будет меняться. Типичная ситуация для гонки данных.

Итак, приложив некоторые усилия, мы прошли два этапа в рассмотренном в п. 1.1 процессе отладки. Наличие ошибки обнаружено, локализация ее также не вызывает сложностей в силу малого размера кода в примере. Осталось понять, в чем же причина ошибки. Но прежде посмотрим, как с ее обнаружением справится ITC.

1.7.3. Подготовка программы для анализа

Выполните следующие действия для подготовки программы к анализу.

1. Верните код примера к исходному состоянию.
2. Конвертируйте проект для использования компилятора Intel® C++ Compiler. В окне **Solution Explorer** выберите файл проекта, щелкните правой кнопкой мыши и в контекстном меню выполните команду **Convert to use Intel® C++ Project System**.
3. В меню **Project** выберите пункт **Properties**, в появившемся окне настроек проекта в дереве слева выберите узел **Configuration Properties→C/C++→General** (рис. 7). В открывшейся таблице справа убедитесь, что значение поля **Debug Information Format** равно **Program Database (/Zi)**.

[3] Конкретное значение в строке «TOTAL = …» может отличаться от указанного.

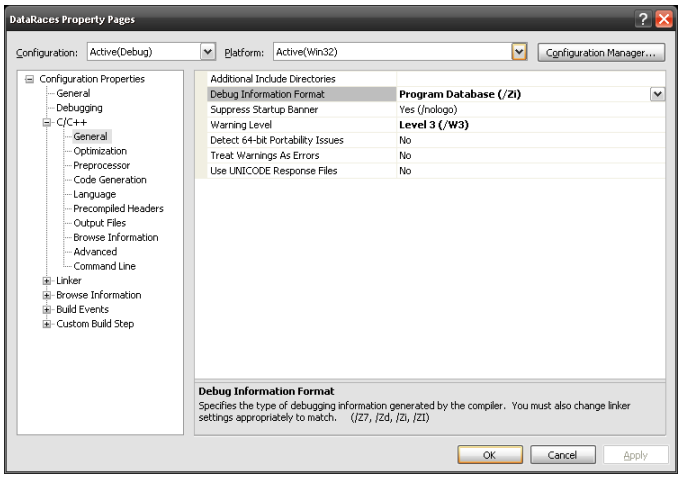

Рис. 7. Указание формата отладочной информации

4. В дереве слева выберите узел **Configuration Properties→C/C++→Optimization** (рис. 8). В открывшейся таблице справа убедитесь, что значение поля **Optimization** равно **Disabled(/Od)**.

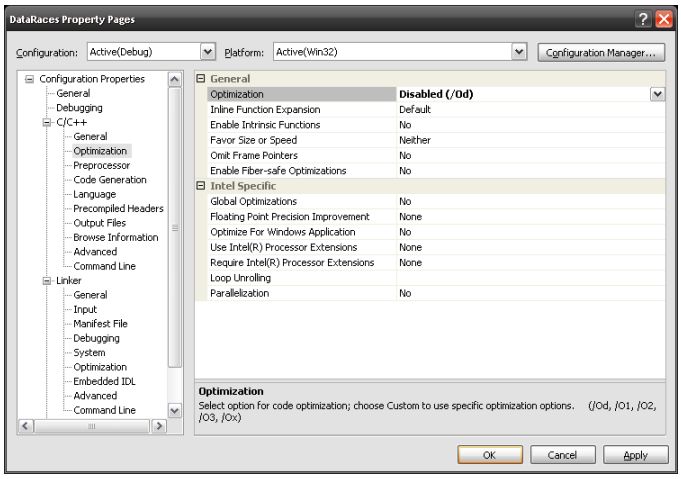

Рис. 8. Отключение оптимизации

5. В дереве слева выберите узел **Configuration Properties→C/C++→Code Generation** (рис. 9). В открывшейся таблице справа убедитесь, что значение поля **Runtime Library** установлено в **Multi-threaded Debug DLL (/MDd)**.

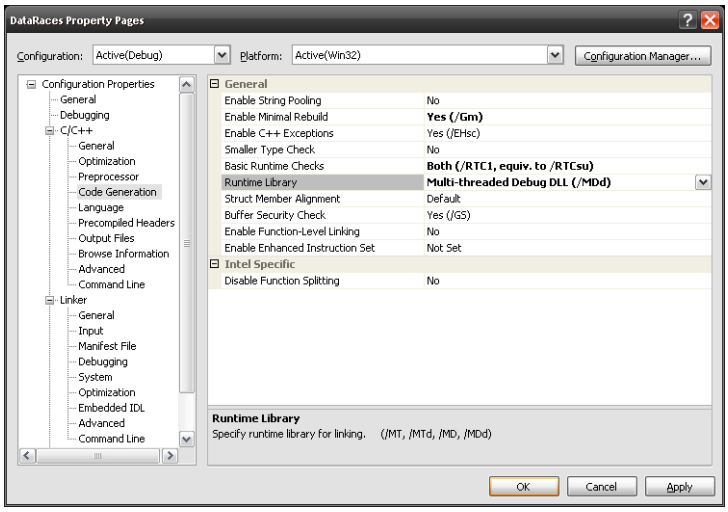

Рис. 9. Выбор потокобезопасных библиотек

6. В дереве слева выберите узел **Configuration Properties→Linker→Command Line** (рис. 10). Убедитесь, что сборка программы выполняется с использованием опции компоновщика **/FIXED:NO**.

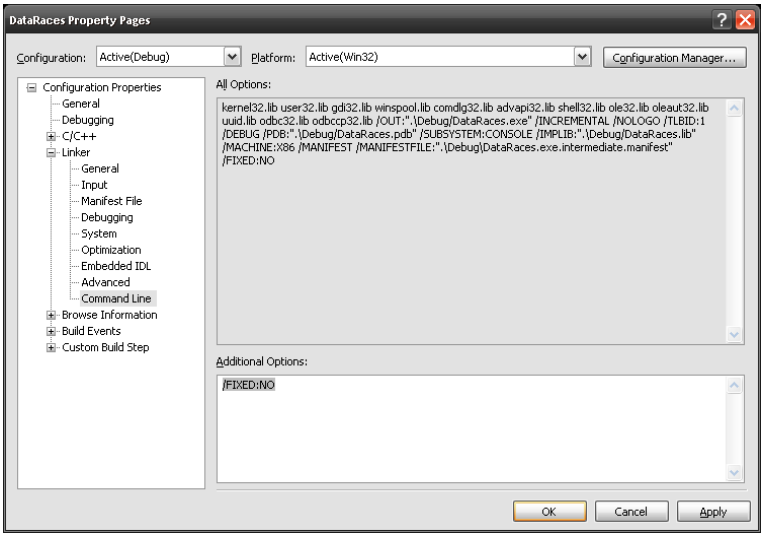

Рис. 10. Установка опций компоновщика

7. Убедитесь, что включена компиляторная инструментация кода. В дереве слева выберите узел **Configuration Properties→C/C++→Command Line** (рис. 11). Убедитесь, что в поле **Additional Options** установлен ключ компилятора **/Qtcheck**.

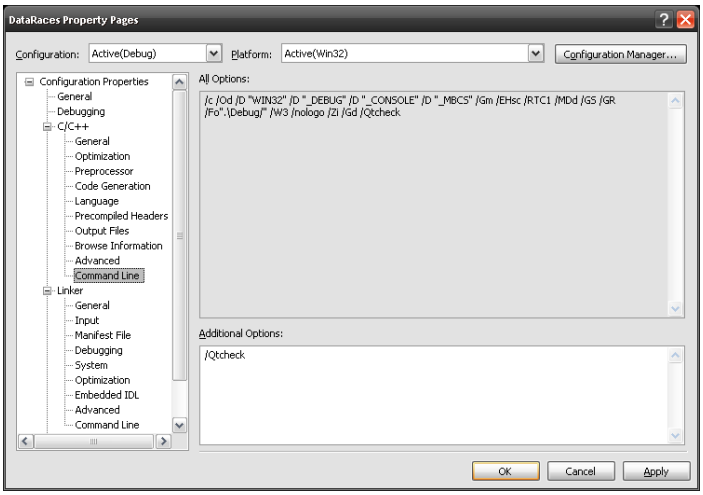

Рис. 11. Установка компиляторного режима инструментации

После выполнения указанных действий программа готова к анализу в ITC.

1.7.4. Создание проекта в Intel Thread Checker

1. Запустите Intel® Thread Checker. Найти его можно, например, по следующему пути: **Start→All programs→Intel(R) Software Development Tools→Intel(R) Thread Checker 3.0→Intel(R) Thread Checker**.
2. В открывшемся окне нажмите на кнопку **New Project** .
3. В окне создания проекта выберите **Intel® Thread Checker Wizard** и нажмите кнопку **OK**.

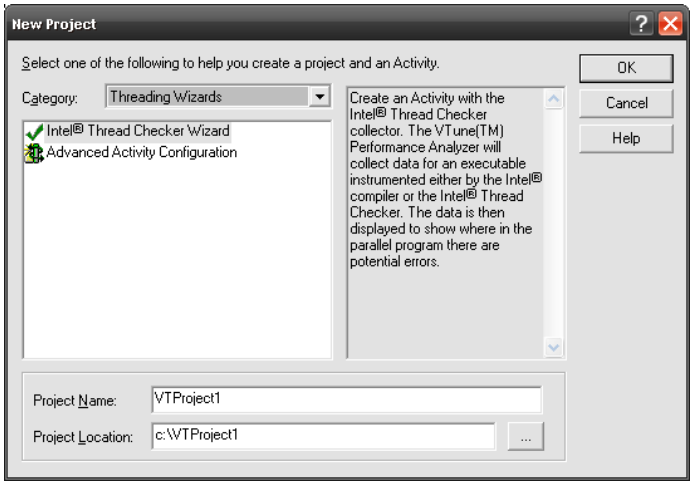

Рис. 12. Выбор типа проекта

4. В окне мастера в поле **Launch an application** укажите путь к исполняемому файлу **C:\ITCLabs\DataRaces\Debug\DataRaces.exe**.

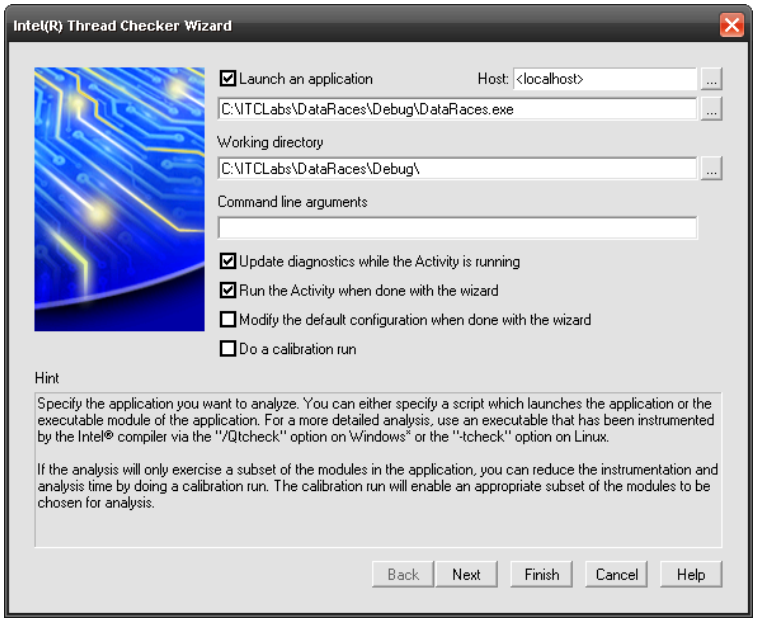

Рис. 13. Выбор программы для анализа

5. Нажмите кнопку **Finish**.

После этого запустится ITC, произведет инструментацию программы и начнет анализ.

1.7.5. Анализ собранной информации

По окончании процесса сбора информации ITC представит результаты в виде, показанном на рис. 14. Итак, мы видим, что ITC нашел в предложенном коде 3 ошибки. При ближайшем рассмотрении видно, что все они указывают на одну и ту же переменную **`globalX`**. Краткие комментарии к диагностикам показывают, что ITC указал все возможные комбинации неверного обращения к переменной **`globalX`**:

- когда первый поток записывает новое значение в переменную **`globalX`**, а второй в это время читает из нее;
- когда первый поток читает из переменной **`globalX`**, а второй в это время в нее пишет;
- когда оба потока одновременно пишут в переменную **`globalX`**.

Несмотря на то, что реально работало 4 потока, очевидно, что для демонстрации ошибки достаточно двух. Именно так ITC всегда и комментирует гонки данных.

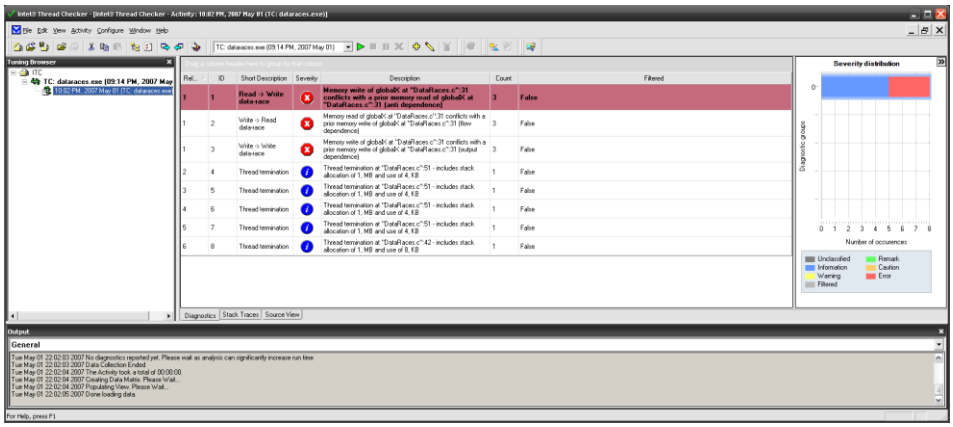

Рис. 14. Результат анализа примера DataRaces – Diagnostics

При наличии отладочной информации ITC может показать в исходном коде местоположение ошибки. Выберите любую из найденных ошибок и двойным щелчком по ней перейдите к просмотру исходного кода (рис. 15).

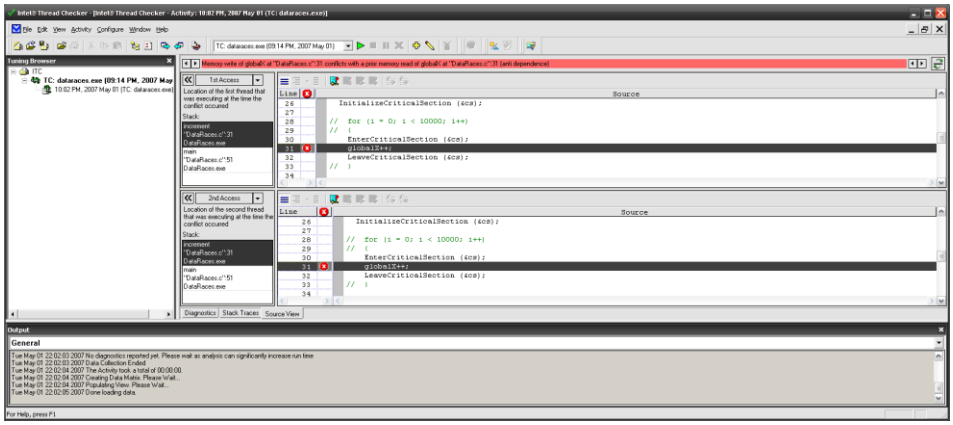

Рис. 15. Результат анализа примера DataRaces – Source View

Комментарий в красном поле над исходными текстами описывает ситуацию. В представленном на рисунке случае имеет место конфликт доступа типа «запись-чтение».

1.7.6. Причина гонки данных и ее устранение

Осталась самая малость – понять, в чем причина гонки данных в рассматриваемом примере, и устранить ее. Сделать это на самом деле не так сложно. Противоречие с тем фактом, что обращение потоков к переменной **`globalX`** происходит внутри критической секции, а гонка данных все равно имеется, кажущееся. Проблема в данном случае не в критической секции, а в объекте, на котором она основана (**`CRITICAL_SECTION cs`**). Этот объект объявлен внутри потоковой функции, а значит, является локальным для каждого потока. То есть когда один поток захва-

тывает критическую секцию, он делает это для своего локального объекта **cs**, к которому остальные потоки не имеют доступа.

Исправить ситуацию можно несколькими способами, но в любом из них объявление объекта **cs** нужно вынести из потоковой функции **increment**, а инициализацию секции и освобождение ресурсов поместить в функцию **main**.

Возможный корректный вариант представлен ниже.

```
#include <stdio.h>
#include <windows.h>

#define NTHREADS 4

int globalX = 0;
CRITICAL_SECTION cs;

DWORD WINAPI increment (void *arg)
{
  EnterCriticalSection (&cs);
    globalX++;
  LeaveCriticalSection (&cs);

  return 0;
}

int main (int argc, char *argv[])
{
  HANDLE h[NTHREADS];
  DWORD  rc;
  int i;

  printf ("START\n");

  InitializeCriticalSection (&cs);

  for (i = 0; i < NTHREADS; i++)
  {
    h[i] = CreateThread (0, 0, increment, NULL, 0, NULL);
  }

  rc = WaitForMultipleObjects (NTHREADS, h, TRUE, INFINITE);

  DeleteCriticalSection (&cs);

  printf ("TOTAL = %d\n", globalX);
  printf ("STOP\n");
}
```

1.8. Литература

Использованные источники

1. Intel® Thread Checker for Windows*. Getting Started Guide. Version 3.0. — Intel Corporation, 2006.

2. Intel® Thread Checker Help. Version 3.0. — Intel Corporation, 2006.
3. Intel® Thread Checker. Guide to Sample Code. Version 3.0. — Intel Corporation, 2006.

Рекомендуемая литература

4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003 (Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley, 2000).
5. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004.

2. Лабораторная работа. Отладка параллельной программы с использованием Intel Thread Checker

2.1. Введение

Говорят: «Любая программа содержит хотя бы одну ошибку». Применительно к параллельному программированию этот тезис можно переформулировать так: «Параллельное программирование начинается с параллельных ошибок». Действительно, процесс создания параллельной программы, либо с нуля на основе параллельного алгоритма, либо путем распараллеливания существующей реализации, вносит в разработку программной системы дополнительные сложности, которые неизбежно приводят к появлению дополнительных по сравнению с последовательным программированием ошибок. Конечно, на основе некоторого опыта можно выделить типовые ошибки, встречающиеся в процессе разработки параллельной программы, и, вооружившись этим знанием, пытаться в дальнейшем не допускать их, однако на практике… Логика параллельных программ существенно более сложна, чем у программ последовательных. Известно, что поиск ошибки начинается с обнаружения ее наличия. В параллельной программе, характер выполнения которой весьма часто носит «налет случайности», ошибка может проявляться один раз на сотню и более запусков. Как «поймать» столь неуловимую вестницу беды в программе? Остро необходима инструментальная поддержка процесса отладки.

В настоящей лабораторной работе рассматривается один из инструментов отладки, основанный на сборе и автоматическом анализе информации по результатам выполнения программ и предназначенный для многопоточных и OpenMP-программ, – Intel® Thread Checker (ITC).

В качестве полигона для демонстрации ошибок и возможностей инструментов по их поиску и анализу используются: классическая задача скалярного умножения векторов; задача Дирихле для уравнения Пуассона, решаемая методом Гаусса–Зейделя; известная задача об обедающих философах; задача «о роботе».

2.2. Методические рекомендации

2.2.1. Цели и задачи работы

Целью данной лабораторной работы является приобретение практических навыков отладки параллельных программ для систем с распределенной памятью, использующих для организации параллелизма либо механизм потоков, либо технологию OpenMP.

Данная цель предполагает решение следующих задач:

- ознакомление с классификацией ошибок, возникающих при разработке многопоточных программ;
- изучение функциональности инструмента отладки Intel Thread Checker;
- изучение ряда учебных примеров, направленных на демонстрацию принципов отладки параллельных программ при помощи инструмента отладки Intel Thread Checker;
- самостоятельная разработка и отладка параллельных программ, использующих Windows Threads или технологию OpenMP.

2.2.2. Структура работы

Данная лабораторная работа состоит из введения, двух разделов, списка дополнительных заданий и списка литературы. Во введении обосновывается актуальность инструментальной поддержки в процессе отладки параллельных программ. В первом разделе приводятся методические рекомендации к лабораторной работе: формулируются цели и задачи, системные требования, рекомендации по проведению занятий. Во втором разделе изучается отладка параллельных программ с демонстрацией характерных ошибок и знакомством с методами их обнаружения и устранения. Изучение проводится на специально подобранных примерах. Для обнаружения ошибок используется инструмент отладки Intel Thread Checker. В заключение приводятся задания для самостоятельной проработки, а также использованная и рекомендуемая литература.

2.2.3. Системные требования

В документации к ITC [2] приводятся следующие системные требования:

Аппаратное обеспечение

Минимальные требования

- процессор Pentium® 4;
- ОЗУ 512 Мб;
- свободное дисковое пространство 300 Мб.

При практическом использовании ITC рекомендуются повышенные требования к минимально необходимым аппаратным ресурсам (для достижения приемлемых показателей оперативности работы):

- процессор Intel® Pentium® 4, поддерживающий технологию Hyper-Threading, или процессор Intel® Xeon®;
- ОЗУ 2 Гб.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

Программное обеспечение

- Microsoft Windows XP Professional или Microsoft Windows Server 2003, или Microsoft Windows XP Professional x64 Edition[4];
- Intel® VTune™ Performance Analyzer версии 7.2 или выше;
- Microsoft Internet Explorer 6.0 или выше;
- Microsoft Visual C++ 6.0 или выше;
- Adobe® Reader®.

Для анализа OpenMP-программ и компиляторной инструментации требуется один из компиляторов:

- Intel® C++ Compiler 8.1 для Windows для архитектуры IA-32, Package ID: w_fc_pc_8.1.023 или выше;
- Intel® C++ Compiler 9.1 для Windows для архитектуры Intel® EM64T;
- Intel® Fortran Compiler для Windows 8.1, Package ID: w_fc_pc_8.1.023 или выше.

2.2.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется придерживаться следующей последовательности изучения материала:

- Рассмотреть классификацию типовых ошибок при разработке программ, использующих потоки, в том числе OpenMP-программ.
- Получить общее представление об инструменте отладки Intel Thread Checker (см.: 1. Intel Thread Checker. Краткое описание).
- Последовательно изучить 4 описанных далее учебных примера, демонстрирующих разные ошибочные ситуации и приемы их диагностики и устранения. При изучении примеров 2–4 по возможности рекомендуется предложить слушателям самостоятельно найти ошибки в приведенных программных реализациях, используя Intel Thread Checker, а затем исправить их.
- Перейти к самостоятельному выполнению слушателями дополнительных заданий.

2.3. Задача о скалярном произведении векторов

2.3.1. Постановка задачи

Задача скалярного умножения векторов отличается простотой математической постановки и очень не сложной программной реализацией (естественно, речь идет о последовательной неоптимизированной версии). Вместе с тем эта задача является алгоритмическим элементом других более трудоемких операций (умножение матрицы на вектор, матричное умножение), да и сама служит хорошим примером, на котором можно демонстрировать большое количество эффектов, появляющихся как при распараллеливании, так и при оптимизации кода.

Итак, пусть имеются векторы a и b размерности n (состоящие из n элементов). Скалярным произведением векторов a и b называется число c, получаемое как

[4] Существует версия ITC и для операционных систем семейства Linux, не имеющая, правда, графического интерфейса и работающая только из командной строки.

$$c = (a,b) = \sum_{j=1}^{n} a_j b_j \ .$$

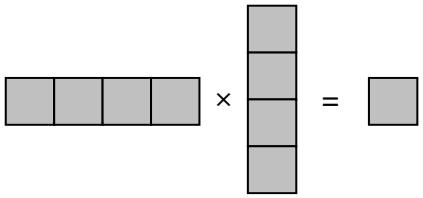

Рис. 1. Скалярное произведение двух векторов

2.3.2. Последовательная реализация

Алгоритм вычисления скалярного произведения полностью описывается его формулой, поэтому приведем сразу его код:

```
sum_all = 0;
for (i = 0; i < Size; i++)
{
  sum_all += x[i] * y[i];
}
```

Необходимые объявления переменных, а также ввод данных здесь рассматривать не будем.

2.3.3. Параллельная реализация 1

Алгоритм скалярного умножения является примером с практически идеальным теоретическим распараллеливанием. Каждая операция умножения компонент векторов может быть выполнена независимо. Единственное «узкое» место – формирование итоговой суммы – может быть легко устранено накоплением результатов в локальных для каждого потока переменных (данных подход демонстрируется в параллельной реализации 2). Однако на практике получить при расчете скалярного произведения хорошие показатели ускорения весьма непросто в силу чрезвычайной легковесности операции в распараллеливаемом цикле. Поскольку в данный момент нас интересует лишь создание работоспособной параллельной версии (с использованием ITC при необходимости), вопросы производительности мы затрагивать не будем.

Итак, мы имеем цикл с регулярной структурой и одинаковой вычислительной сложностью итераций. Распараллеливание его средствами OpenMP не представляет никакого труда.

```
sum_all = 0;
#pragma omp parallel for
for (i = 0; i < N; i++)
{
  sum_all += up[i] * vp[i];
}
```

Некоторый произвол возможен в том, как разделить вычисления между потоками. Первая схема – каждый поток обрабатывает непрерывный фрагмент умно-

жаемых векторов. Вторая – векторы делятся между потоками поэлементно (или поблочно). Каждая схема может быть реализована с использованием параметра `schedule`. Провести соответствующие эксперименты предоставляем читателю самостоятельно.

В представленном выше коде мы сознательно допустили достаточно очевидную ошибку. Посмотрим, как с ней справится ITC.

2.3.4. Анализ параллельной реализации 1

Откройте проект **Scalar**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005;**
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ITCLabs\Scalar**;
- дважды щелкните на файле **Scalar.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **Scalar.cpp**. После этих действий программный код, с которым предстоит работать, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

Далее выполните действия, описанные в пункте 1.7.3, чтобы подготовить программу для анализа в ITC.

И, наконец, запустите ITC, создайте в нем новый проект и укажите путь к исполняемому файлу **C:\ITCLabs\Scalar\Debug\Scalar.exe**.

Нажмите кнопку **Finish**. После этого запустится ITC, произведет инструментацию программы и начнет анализ. Результатом станет появление следующей диагностики.

1	**1**	**Read -> Write data-race**		**Memory write of sum_all at "ompScalar.cpp":24 conflicts with a prior memory read of sum_all at "ompScalar.cpp":24 (anti dependence)**	**99**
1	2	Write -> Read data-race		Memory read of sum_all at "ompScalar.cpp":24 conflicts with a prior memory write of sum_all at "ompScalar.cpp":24 (flow dependence)	99
1	3	Write -> Write data-race		Memory write of sum_all at "ompScalar.cpp":24 conflicts with a prior memory write of sum_all at "ompScalar.cpp":24 (output dependence)	99
2	4	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 2. Результат анализа параллельной реализации 1 скалярного умножения

Как и следовало ожидать, источник неприятностей – переменная `sum_all`, на что недвусмысленно указывает ITC, сигнализируя о гонке данных при работе с этой переменной.

Для исправления представленного выше кода требуется лишь одна модификация – переменная, в которую накапливается сумма, должна быть сделана локальной для каждого потока. В противном случае во избежание конфликта доступа придется заключать операцию суммирования в критическую секцию, что сведет эффект от распараллеливания к нулю. Естественно, по окончании работы потоков необходимо собрать из локальных сумм общий результат. Реализовать эту схему в

OpenMP можно различными способами; рассмотрим наиболее быстрый вариант, использующий параметр **`reduction`** директивы **`parallel for`**.

Вносим исправления:

```
sum_all = 0;
#pragma omp parallel for reduction(+:sum_all)
for (i = 0; i < N; i++)
{
  sum_all += up[i] * vp[i];
}
```

Повторно запускаем активность в ITC и убеждаемся в отсутствии ошибок.

Re...	ID	Short Description	Severity	Description	Count
1	1	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 3. Результат анализа параллельной реализации 1 после исправления

Отметим, что параметр **`reduction(+:sum_all)`** решает обе заявленные выше задачи (локализации переменной суммирования и подсчета итоговой суммы) и это наиболее короткий из возможных вариантов.

2.3.5. Параллельная реализация 2

В дополнение рассмотрим еще один, несколько искусственный, вариант скалярного умножения, который, тем не менее, позволит нам продемонстрировать некоторые типичные ошибки, нередко допускаемые при создании OpenMP-программ.

Итак, пусть параллельная схема вычисления скалярного произведения выглядит следующим образом.

```
#pragma omp parallel private(rank, sum)
{
  rank = omp_get_thread_num();
  if (rank == 0)
  {
    NumThreads = omp_get_num_threads();
    x = CreateVector(N);
    y = CreateVector(N);
  }
  sum[rank] = 0;
  #pragma omp for
  for (i = 0; i < N; i++)
  {
    sum[rank] += x[i] * y[i];
  }
}
sum_all = 0;
for (i = 0; i < NumThreads; i++)
{
  sum_all += sum[i];
}

DeleteVector(x);
DeleteVector(y);
```

Нулевой поток «занимается» выделением памяти под данные и их инициализацией, после чего начинается собственно расчет. Представленная схема хотя и выглядит для задачи скалярного умножения чересчур усложненной, но в других ситуациях вполне может с успехом применяться. Отметим также, что здесь использовано другое решение для локализации суммы в потоках – массив **`sum`**.

Попробуйте самостоятельно найти в представленном коде ошибки, прежде чем мы перейдем к анализу.

2.3.6. Анализ параллельной реализации 2

Откройте проект **Scalar2**. Соберите программу в конфигурации **debug**. Прежде всего, посмотрим на результаты работы последовательной и параллельной версий. Зададим размер векторов, равный 100, и запустим программу.

```
Sequential Time: 0.000117100
Sequential scalar product result is 328350.0

Parallel Time: 0.004261040
Parallel scalar product result is -92559631349317831000000000000000000000000000000000000000000000000000.0
```

Рис. 4. Результат запуска параллельной реализации 2

Итак, результаты запуска однозначно указывают на некорректность параллельной версии. Приступаем к поиску ошибок.

Снова выполните действия, описанные в пункте 1.7.3, чтобы подготовить программу для анализа в ITC. Запустите ITC, создайте в нем новый проект и укажите путь к исполняемому файлу **C:\ITCLabs\Scalar2\Debug\Scalar2.exe**.

Нажмите кнопку **Finish**. После этого запустится ITC, произведет инструментацию программы и начнет анализ. Результатом станет появление следующей диагностики:

1	1	OpenMP - use of unsupported...		OpenMP - use of unsupported API at "ompScalar.cpp":60	1
1	2	Write -> Write data-race		Memory write of NumThreads at "ompScalar.cpp":63 conflicts with a prior memory write of NumThreads at "ompScalar.cpp":63 (output dependence)	1
1	3	Write -> Write data-race		Memory write of x at "ompScalar.cpp":64 conflicts with a prior memory write of x at "ompScalar.cpp":64 (output dependence)	1
1	4	Write -> Write data-race		Memory write of y at "ompScalar.cpp":65 conflicts with a prior memory write of y at "ompScalar.cpp":65 (output dependence)	1
1	5	Write -> Read data-race		Memory read of x at "ompScalar.cpp":71 conflicts with a prior memory write of x at "ompScalar.cpp":64 (flow dependence)	100
1	6	Read -> Write data-race		Memory write of x at "ompScalar.cpp":64 conflicts with a prior memory read of x at "ompScalar.cpp":71 (anti dependence)	100
1	7	Write -> Read data-race		Memory read of y at "ompScalar.cpp":71 conflicts with a prior memory write of y at "ompScalar.cpp":65 (flow dependence)	100
1	8	Read -> Write data-race		Memory write of y at "ompScalar.cpp":65 conflicts with a prior memory read of y at "ompScalar.cpp":71 (anti dependence)	100
2	9	OpenMP -- cannot be private		OpenMP -- The access at "ompScalar.cpp":71 cannot be private because it expects the value previously defined at "ompScalar.cpp":71 in the serial execution	99
3	10	OpenMP -- undefined in the serial code (original program)		OpenMP -- undefined in the serial code (original program) at "ompScalar.cpp":77 with "ompScalar.cpp":71	1
3	11	OpenMP -- undefined in the serial code (original program)		OpenMP -- undefined in the serial code (original program) at "ompScalar.cpp":77 with "ompScalar.cpp":50	1
4	12	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 5. Результат анализа параллельной реализации 2 скалярного умножения

Первое, что можно отметить, – появились новые типы сообщений. Вернемся к этому позже. Второй интересный момент – ошибка под номером 1.

«Открываем» ее двойным щелчком.

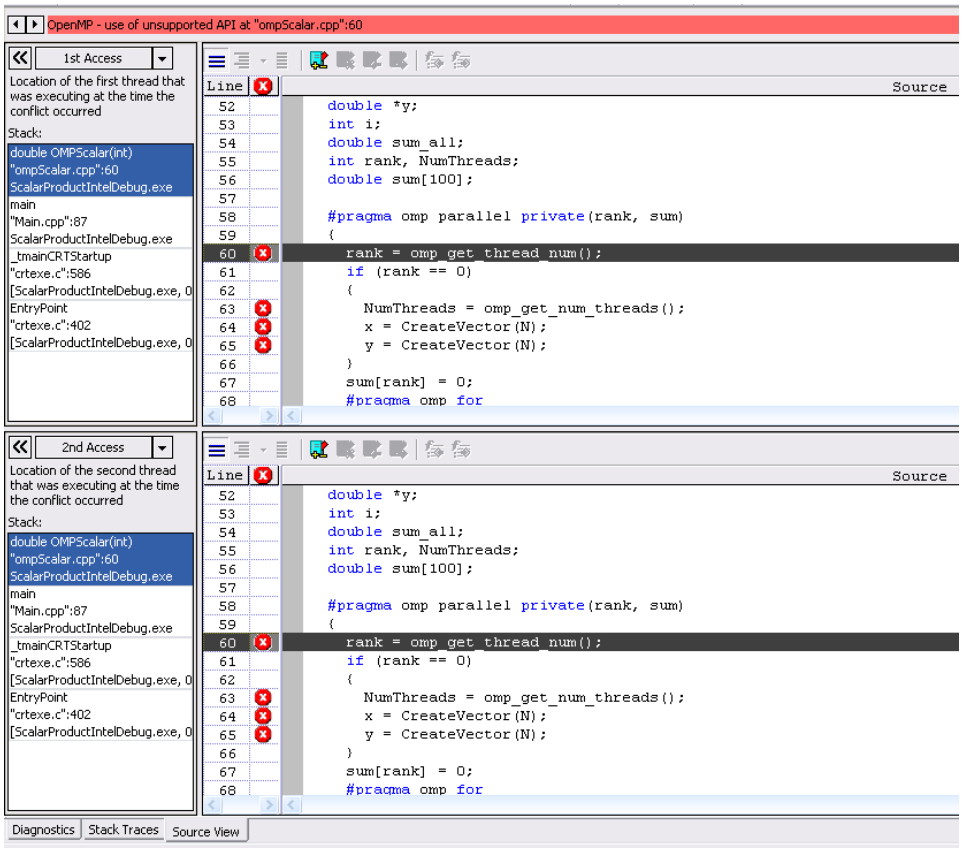

Рис. 6. Просмотр диагностики «Use of unsupported API»

Представленная диагностика вызвана тем, что ITC «не понимает» run-time-функции OpenMP, и в частности **`omp_get_num_threads()`**. Отсюда, кстати, и большая часть остальных ошибок, которые вызваны тем, что значение переменной **`rank`** неизвестно ITC, а соответственно, он находит конфликты там, где их на самом деле нет.

Выходов из подобных ситуаций два: отказаться от использования run-time-функций OpenMP (тем более что их использование при программировании на OpenMP считается чем-то вроде плохого тона) или отказаться от компиляторного инструментирования, то есть удалить из опций проекта ключ /Qtcheck.

Но прежде вернемся к ошибке под номером 9.

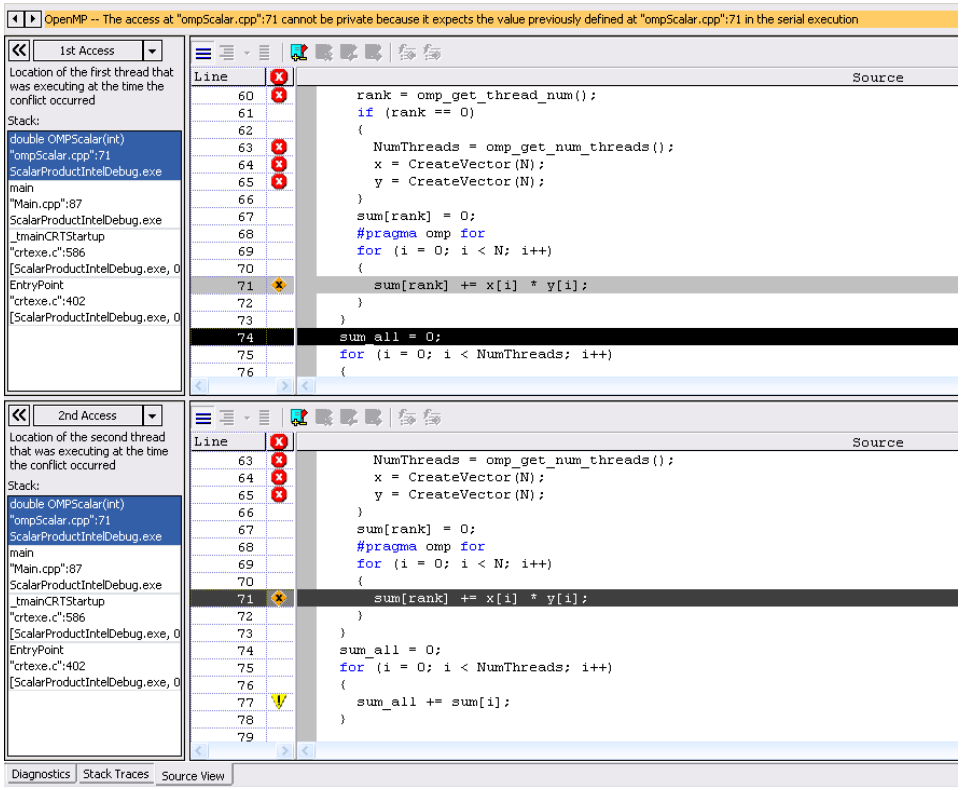

Рис. 7. Неверная локализация переменной sum

Несмотря на не слишком вразумительное сообщение, тем не менее, можно понять, что выше по коду нами допущена ошибка – переменная **sum** сделана **private**, что, конечно же, неверно. Этот массив специально был создан для подсчета потоками локальных сумм и, естественно, должен быть общим, чтобы по окончании параллельной секции из него в переменную **sum_all** можно было собрать итоговую сумму.

Исправляем ситуацию:

```
#pragma omp parallel private(rank/*, sum*/)
{
  ...
}
sum_all = 0;
for (i = 0; i < NumThreads; i++)
{
  sum_all += sum[i];
}

DeleteVector(x);
DeleteVector(y);
```

Убираем ключ /Qtcheck из проекта и снова запускаем анализ в ITC.

Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Read data-race**		**Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "ompScalar.cpp":103 (flo...**	**50**
1	2	Write -> Read data-race		Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "Vector.cpp":29 (flow dependence)	50
1	3	Write -> Read data-race		Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "ompScalar.cpp":104 (flow dependence)	50
1	4	Write -> Read data-race		Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "Vector.cpp":29 (flow dependence)	50
2	5	Thread termination		Thread termination at "ompScalar.cpp":97 - includes stack allocation of 3, MB and use of 4, KB	1
3	6	Thread termination		Thread termination at "ompScalar.cpp":97 - includes stack allocation of 1, MB and use of 4, KB	1
4	7	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 8. Результат анализа параллельной реализации 2 после первого исправления

Теперь мы, к сожалению, потеряли информацию об именах переменных. Число сообщений также сократилось. Однако ITC все еще видит в коде ошибки.

Анализируя первое же сообщение, мы находим одну из типичных ошибок в многопоточных программах – использование потоками неинициализированных данных.

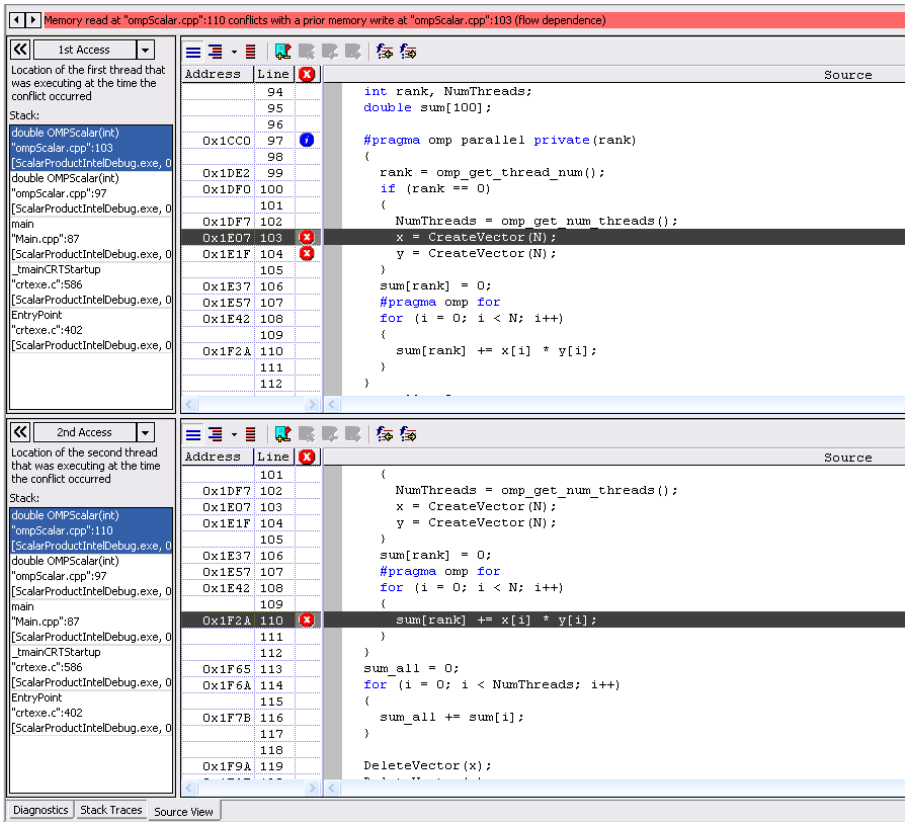

Рис. 9. Ошибка – использование потоками неинициализированных данных

Глядя на выделенные строки, нетрудно понять, что поток с номером, отличным от 0, будет пытаться использовать переменные **x** и **y** до того, как нулевой поток выделит под них память, что в лучшем случае приведет к падению, а в худшем к неверному результату расчетов. Решение в данной ситуации также является типичным и состоит в установке барьера после участка инициализации.

```
#pragma omp parallel private(rank)
{
  rank = omp_get_thread_num();
  if (rank == 0)
  {
    NumThreads = omp_get_num_threads();
    x = CreateVector(N);
    y = CreateVector(N);
  }
  sum[rank] = 0;
  #pragma omp barrier
  #pragma omp for
  for (i = 0; i < N; i++)
  {
    sum[rank] += x[i] * y[i];
  }
}
...
```

После внесения последних исправлений результаты запуска полученной параллельной версии, наконец-то, совпадут с последовательным вариантом. Убедитесь в этом, а также в том, что ITC подтвердит отсутствие проблем в коде.

2.4. Задача Дирихле

2.4.1. Постановка задачи

В качестве второго примера рассмотрим задачу из области численного решения дифференциальных уравнений в частных производных, а именно задачу Дирихле для уравнения Пуассона (см., например, [8]). Итак, необходимо найти функцию $u = u(x, y)$, удовлетворяющую в области определения D уравнению $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \quad (x, y) \in D,$ и принимающую на границе D^0 области D значения $g(x, y)$: $u(x, y) = g(x, y), \ (x, y) \in D^0$,

2.4.2. Метод решения

Используя распространенный метод конечных разностей (он же метод сеток), перепишем уравнение Пуассона в конечно-разностной форме [9]

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

Разрешив его относительно u_{ij}, получим

$$u_{ij} = 0.25 * (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Мы получили основу для построения итерационной схемы решения задачи Дирихле, в которой, отталкиваясь от некоторого начального приближения, можно последовательно уточнять значения u_{ij} до достижения требуемой точности.

2.4.3. Последовательная реализация

Последовательная реализация данной итерационной схемы может выглядеть следующим образом:

```
do
{
  dmax = 0;
  for (i = 1; i < N - 1; i++)
  {
    for(j = 1; j < N - 1; j++)
    {
      temp = u[N * i + j];
      u[N * i + j] = 0.25 * (u[N * i + j + 1] + u[N * i + j - 1] +
        u[N * (i + 1) + j] + u[N * (i - 1) + j]);
      dm = fabs(u[N * i + j] - temp);
      if (dmax < dm)
        dmax = dm;
    }
  }
}
while (dmax > EPS);
```

Здесь **dmax** есть максимальная разность между «старым», с предыдущей итерации, и «новым», посчитанным на текущей, значениями u_{ij}, которая используется для принятия решения об окончании расчетов.

2.4.4. Параллельная реализация 1

«Сеточные» задачи, в которых решение получается выполнением набора из одних и тех же действий в каждом «узле» сетки, обладают очень простой схемой распараллеливания. Достаточно сетку «разрезать» на части: вертикально на столбцы, горизонтально на полосы или и так и так, то есть на блоки, – и распараллеливание выполнено. Если при этом объем вычислений, выполняемых в каждом узле, примерно одинаков, то и эффективность полученной параллельной реализации может быть довольно высока.

В данной работе мы рассмотрим две схемы распараллеливания решения задачи Дирихле. Первая – схема с максимальным параллелизмом, в которой сетка «режется» на блоки размера 1 на 1. Вторая – схема, в которой распределение вычислений осуществляется по строкам сетки.

Нашей целью не является производительность получаемых параллельных реализаций, поэтому в качестве первого варианта, как было сказано выше, мы используем вариант с максимальным параллелизмом, который в OpenMP-версии достигается распараллеливанием обоих циклов:

```
do
{
  dmax = 0;
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++)
  {
    #pragma omp parallel for
    for(j = 1; j < N - 1; j++)
    {
      temp = u[N * i + j];
      u[N * i + j] = 0.25 * (u[N * i + j + 1] + u[N * i + j - 1] +
        u[N * (i + 1) + j] + u[N * (i - 1) + j]);
      dm = fabs(u[N * i + j] - temp);
      if (dmax < dm)
        dmax = dm;
    }
  }
}
while (dmax > EPS);
```

Конечно же, в представленном виде код является некорректным. В чем именно – нам предстоит выяснить, используя Intel Thread Checker.

2.4.5. Анализ параллельной реализации 1

Исследуем представленный выше код на наличие ошибок.

Для проверки правильности работы параллельной программы можно применить, казалось бы, естественный подход – сравнить результаты выполнения последовательной и параллельной версий. Например, в задаче умножения матриц результат должен быть один и тот же, независимо от применяемой вычислительной схемы. Вместе с тем часто наблюдается и иная картина – именно так, как в рассматриваемом примере – результаты параллельной версии могут отличаться от результатов последовательной, но это не будет являться признаком наличия ошибок. Одна из причин различия результатов – изменение порядка выполнения операций вещественной арифметики, что может, в частности, привести к изменению величины получаемой погрешности вычислений. Ситуация усложняется, если вычислительная схема параллельного алгоритма отличается от исходного последовательного прототипа. Так, в нашем примере порядок обработки узлов вычислительной сетки в параллельной версии алгоритма может быть другим, нежели в последовательном методе. Более того, результаты параллельных расчетов могут не совпадать при различных запусках даже при одних и тех же начальных данных, поскольку условия запуска (например, загрузка вычислительной системы) также могут влиять на порядок вычислений. Описанная проблема – различие выполняемых вычислений – является одной из принципиальных при разработке параллельных программ. Наличие такой проблемы приводит к тому, что при разработке параллельного метода необходимо тщательно анализировать идентичность вычислительных схем параллельных и последовательных расчетов, а при обнаружении различий доказывать корректность параллельной версии и определять способы проверки правильности выполнения параллельных расчетов (дополнительная информация по данному вопросу может быть получена, например, в материале лекции 11 в [7]).

Рассматриваемый пример прекрасно демонстрирует описанные выше ситуации – вычислительная схема параллельного алгоритма отличается от последовательного метода (порядки обработки узлов вычислительной сетки могут оказаться различными) и результаты параллельных вычислений могут отличаться от запуска к запуску. В теоретических работах показана корректность рассмотренной параллельной схемы вычислений – процесс вычислений сходится к решению поставленной задачи Дирихле. Данный математический результат дает подход для проверки правильности выполнения параллельной программы путем оценки точности получения требуемого результата вычислений (в силу сходимости параллельный алгоритм должен обеспечивать достижение любой наперед заданной точности расчетов). Иными словами, для проверки правильности можно сравнить результат параллельной программы с точным решением, и если результат совпадает с точным решением в пределах задаваемой точности, то можно считать, что расчеты выполнены правильно. При этом, однако, следует понимать, что такая проверка не дает полной гарантии – как и любое тестирование, правильность прохождения теста говорит о корректности работы программы только при данных исходных параметрах задачи (а в случае параллельных вычислений – о корректности только данного конкретного запуска программы). Необходимы многократные запуски при одних и тех же исходных данных, нужны различные варианты постановок решаемой задачи. Процесс тестирования становится трудоемким и длительным – все отмеченные моменты убедительно подтверждают необходимость использования инструментов отладки параллельных программ.

Проанализируем код с помощью Intel Thread Checker.

Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Write data-race**		**Memory write of temp at "ompGZ.cpp":39 conflicts with a prior memory write of temp at "ompGZ.cpp":39 (output dependence)**	**20273**
1	2	Read -> Write data-race		Memory write of temp at "ompGZ.cpp":39 conflicts with a prior memory read of temp at "ompGZ.cpp":42 (anti...	1986754
1	3	Write -> Read data-race		Memory read of v[] at "ompGZ.cpp":40 conflicts with a prior memory write of v[] at "ompGZ.cpp":40 (flow dependence)	1986754
1	4	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":40 conflicts with a prior memory read of v[] at "ompGZ.cpp":40 (anti dependence)	1986754
1	5	Write -> Write data-race		Memory write of dm at "ompGZ.cpp":42 conflicts with a prior memory write of dm at "ompGZ.cpp":42 (output...	20273
1	6	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":42 conflicts with a prior memory read of dm at "ompGZ.cpp":45 (anti dependence)	9506
1	7	Write -> Read data-race		Memory read of dmax at "ompGZ.cpp":44 conflicts with a prior memory write of dmax at "ompGZ.cpp":45 (flow...	1846285
1	8	Write -> Write data-race		Memory write of dmax at "ompGZ.cpp":45 conflicts with a prior memory write of dmax at "ompGZ.cpp":45 (output dependence)	3001
1	9	Read -> Write data-race		Memory write of dmax at "ompGZ.cpp":45 conflicts with a prior memory read of dmax at "ompGZ.cpp":44 (anti...	16309
1	10	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":42 conflicts with a prior memory read of dm at "ompGZ.cpp":44 (anti dependence)	1977248
2	11	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 10. Задача Дирихле. Результат анализа параллельной реализации 1

Как видим, налицо существенное количество ошибок, которые ITC классифицирует как гонку данных.

«Разворачивая» любую из них, мы получим информацию, аналогичную представленной на рис. 11.

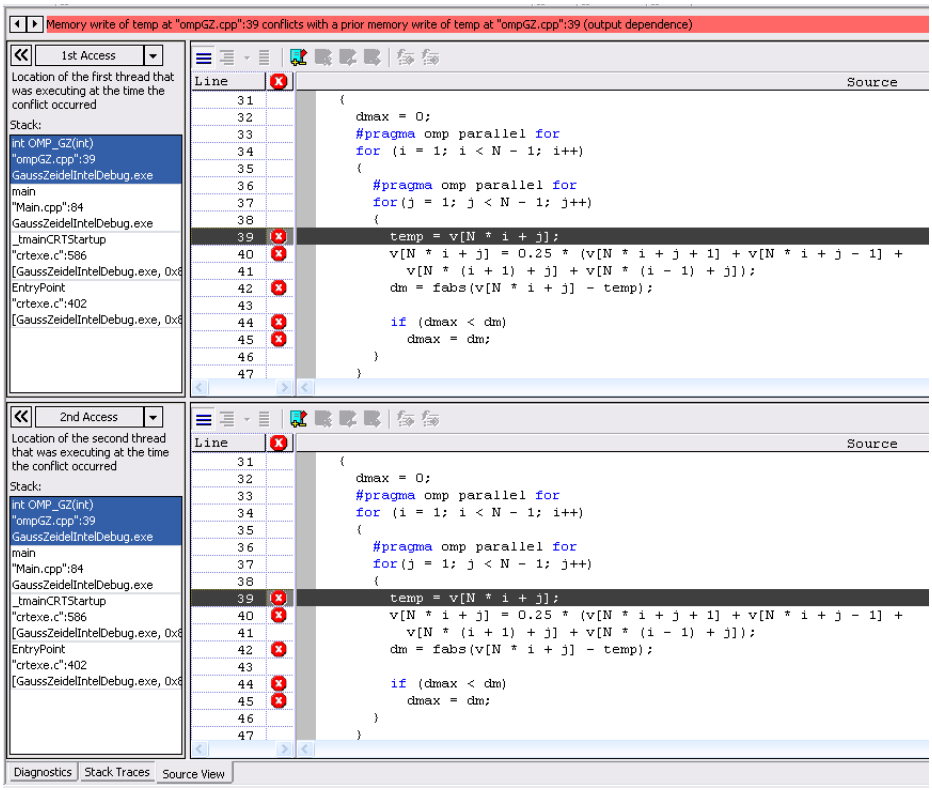

Рис. 11. Задача Дирихле. Ошибка типа «гонка данных»

Как видим, диагностика ITC указывает нам и строку, в которой имеет место проблема, и переменную, с которой она связана. В выделенной строке обнаружен конфликт доступа к переменной **temp**, когда два потока могут пытаться одновременно записать в нее значения.

Решение выявленных проблем зависит от того, как используется та или иная переменная, и может состоять в ее «локализации», то есть в создании внутренней для каждого потока копии, как для переменной **temp**, или в синхронизации доступа к ней, если переменная нужна всем потокам, как в случае с **dmax**.

Заметим, что две из найденных ITC ошибок в данной реализации могут проявиться только при очень маленьком значении *N*. Какие именно – предлагаем читателям найти самостоятельно.

Корректируем код:

```
omp_lock_t dmax_lock;

omp_init_lock(&dmax_lock);
do
{
  dmax = 0;
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++)
  {
    #pragma omp parallel for private(temp, dm)
```

```
    for(j = 1; j < N - 1; j++)
    {
      temp = v[N * i + j];
      v[N * i + j] = 0.25 * (v[N * i + j + 1] + v[N * i + j - 1] +
        v[N * (i + 1) + j] + v[N * (i - 1) + j]);
      dm = fabs(v[N * i + j] - temp);

      omp_set_lock(&dmax_lock);
      if (dmax < dm)
        dmax = dm;
      omp_unset_lock(&dmax_lock);
    }
  }
}
while (dmax > EPS);

omp_destroy_lock(&dmax_lock);
```

По результатам запуска убеждаемся, что параллельная версия работает корректно. Снова запускаем Thread Checker и получаем новые данные (рис. 12).

Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Read data-race**		**Memory read of v[] at "ompGZ.cpp":138 conflicts with a prior memory write of v[] at...**	**198675**
1	2	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":138 conflicts with a prior memory read of v[] at "ompGZ.cpp":138 (anti dependence)	1986754
2	3	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 12. Задача Дирихле. Результат анализа параллельной реализации 1 после исправления

Прав или нет ITC в своих подозрениях относительно последнего варианта кода – предлагаем читателям выяснить самостоятельно.

2.4.6. Параллельная реализация 2

Второй вариант распараллеливания, который мы будем использовать в данной работе, состоит в уменьшении степени параллелизма реализации, что, тем не менее, ведет к увеличению эффективности. Вариант заключается в распараллеливании только внешнего цикла и дополнительно предполагает некоторые изменения в схеме подсчета значения **dmax**:

```
do
{
  dmax = 0;
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++)
  {
    dm = 0;
    for(j = 1; j < N - 1; j++)
    {
      temp = u[N * i + j];
      u[N * i + j] = 0.25 * (u[N * i + j + 1] + u[N * i + j - 1] +
        u[N * (i + 1) + j] + u[N * (i - 1) + j]);
```

```
      d = fabs(u[N * i + j] - temp);
      if (dm < d)
        dm = d;
    }
    if (dmax < dm)
      dmax = dm;
  }
}
while (dmax > EPS);
```

Как и в первом случае, представленный код содержит ошибки, которые необходимо найти с помощью Intel Thread Checker и устранить.

2.4.7. Анализ параллельной реализации 2

Прежде всего, отметим, что в связи с изменившейся схемой работы и рассмотренная выше параллельная версия и данная могут не давать идентичные результаты по сравнению с последовательным кодом, что, конечно, является дополнительным усложняющим моментом для «ручной» отладки.

Проанализируем код с помощью Intel Thread Checker.

Relat...	ID	Short Description	Severity	Description	Count
1	1	Write -> Write data-race		Memory write of dm at "ompGZ.cpp":182 conflicts with a prior memory write of dm at "ompGZ.cpp":190 (output dependence)	20273
1	2	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":182 conflicts with a prior memory read of dm at "ompGZ.cpp":193 (anti dependence)	20273
1	3	Write -> Write data-race		Memory write of j at "ompGZ.cpp":183 conflicts with a prior memory write of j at "ompGZ.cpp":183 (output dependence)	20273
1	4	Read -> Write data-race		Memory write of j at "ompGZ.cpp":183 conflicts with a prior memory read of j at "ompGZ.cpp":183 (anti dependence)	40546
1	5	Write -> Write data-race		Memory write of temp at "ompGZ.cpp":185 conflicts with a prior memory write of temp at "ompGZ.cpp":185 (output dependence)	20273
1	6	Read -> Write data-race		Memory write of temp at "ompGZ.cpp":185 conflicts with a prior memory read of temp at "ompGZ.cpp":188 (anti dependence)	1986754
1	7	Write -> Read data-race		Memory read of v[] at "ompGZ.cpp":186 conflicts with a prior memory write of v[] at "ompGZ.cpp":186 (flow dependence)	1986754
1	8	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":186 conflicts with a prior memory read of v[] at "ompGZ.cpp":186 (anti dependence)	1986754
1	9	Write -> Write data-race		Memory write of d at "ompGZ.cpp":188 conflicts with a prior memory write of d at "ompGZ.cpp":188 (output...	20273
1	**10**	**Read -> Write data-race**		**Memory write of d at "ompGZ.cpp":188 conflicts with a prior memory read of d at "ompGZ.cpp":190...**	**9506**
1	11	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":190 conflicts with a prior memory read of dm at "ompGZ.cpp":193 (anti dependence)	334521
1	12	Write -> Read data-race		Memory read of dmax at "ompGZ.cpp":192 conflicts with a prior memory write of dmax at "ompGZ.cpp":193 (flow dependence)	20273
1	13	Write -> Write data-race		Memory write of dmax at "ompGZ.cpp":193 conflicts with a prior memory write of dmax at "ompGZ.cpp":193 (output dependence)	3428
1	14	Read -> Write data-race		Memory write of dmax at "ompGZ.cpp":193 conflicts with a prior memory read of dmax at "ompGZ.cpp":192 (anti dependence)	3428
1	15	Read -> Write data-race		Memory write of d at "ompGZ.cpp":188 conflicts with a prior memory read of d at "ompGZ.cpp":189 (anti dependence)	1977248
2	16	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 13. Задача Дирихле. Результат анализа параллельной реализации 2

Количество ошибок в этой версии еще больше в связи с увеличившимся количеством переменных, кроме того, в список попала переменная **j**, заботу о которой раньше «брал на себя» компилятор, – как известно, переменную цикла в директиве **#pragma omp parallel for** не обязательно объявлять как **private**.

Так же как и для варианта 1, исправление найденных ошибок заключается в локализации необходимых переменных и использовании синхронизации при работе с переменной **dmax**.

```
omp_lock_t dmax_lock;

omp_init_lock(&dmax_lock);
do
{
  dmax = 0;
  #pragma omp parallel for private(j, temp, d, dm)
  for (i = 1; i < N - 1; i++)
  {
    dm = 0;
    for(j = 1; j < N - 1; j++)
    {
      temp = v[N * i + j];
      v[N * i + j] = 0.25 * (v[N * i + j + 1] + v[N * i + j - 1] +
        v[N * (i + 1) + j] + v[N * (i - 1) + j]);
      d = fabs(u[N * i + j] - temp);
      if (dm < d)
        dm = d;
    }
    omp_set_lock(&dmax_lock);
    if (dmax < dm)
      dmax = dm;
    omp_unset_lock(&dmax_lock);
  }
}
while (dmax > EPS);

omp_destroy_lock(&dmax_lock);
```

Снова запускаем ITC. Результат представлен на рис. 14.

Re...	ID	Short Description	Severity	Description	Count
1	1	Write -> Read data-race	✖	Memory read of v[] at "ompGZ.cpp":233 conflicts with a prior memory write of v[] at...	198675
1	2	Read -> Write data-race	✖	Memory write of v[] at "ompGZ.cpp":233 conflicts with a prior memory read of v[] at "ompGZ.cpp":233 (anti dependence)	1986754
2	3	Thread termination	ℹ	Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 14. Задача Дирихле. Результат анализа параллельной реализации 2 после исправления

Как и выше, ITC нашел проблему там, где ее на самом деле нет. Почему это так – предлагаем читателям выяснить самостоятельно. В качестве подсказки: вспомните характер распределения итераций цикла **for** при использовании директивы **#pragma omp parallel for** без дополнительных параметров.

2.5. Задача об обедающих философах

На примере данной классической задачи мы продемонстрируем еще одну типичную ошибку – *тупик* (*deadlock*), ее диагностику при помощи ITC и один из способов устранения.

2.5.1. Постановка задачи

Приведем вольную формулировку задачи Дейкстры: за круглым столом заседают 5 философов. Напротив каждого из них стоит блюдо со спагетти. Между каждыми двумя соседями расположена одна вилка. Философ может находиться в одном из двух состояний: ест, размышляет. При еде философу нужны 2 вилки (левая и правая). Стратегия поведения философа следующая: он берет левую вилку (если она свободна) и затем, дождавшись правой вилки, начинает есть. Поев, он освобождает вилки в обратном порядке. Реализовать симулятор, демонстрирующий заседание философов.

2.5.2. Параллельная реализация 1

Реализуем требуемый симулятор на основе потоков Windows Threads.

Идея реализации состоит в следующем: главный поток создает дополнительные потоки в соответствии с количеством философов, запускает их и переходит в бесконечный цикл. Каждый из дополнительных потоков реализует поведение философа. При этом вилки предлагается моделировать при помощи мьютексов, обеспечивая тем самым процедуру «захвата вилки» философом. Функция потока будет принимать в качестве параметров структуру, содержащую мьютексы, соответствующие левой и правой вилкам, а также порядковый номер философа.

Сделаем следующие объявления:

```
// Количество философов
const unsigned int n = 5;

// Структура – описание философа
typedef struct
{
  int iID;                                  // Номер философа
  HANDLE hMyObjects[2];                     // Мьютексы (вилки)
} THREADCONTROLBLOCK, *PTHREADCONTROLBLOCK;
```

Приведем функцию потока. Используем функцию **WaitForSingleObject** для ожидания освобождения ресурса (мьютекса).

```
long WINAPI ThreadRoutine(long lParam)
{
  PTHREADCONTROLBLOCK pcb=(PTHREADCONTROLBLOCK)lParam;
  while (TRUE)
  {
    WaitForSingleObject(pcb->hMyObjects[0],INFINITE);
    WaitForSingleObject(pcb->hMyObjects[1],INFINITE);
    printf("Eating: Philosopher %d \n",pcb->iID);
    ReleaseMutex(pcb->hMyObjects[1]);
    ReleaseMutex(pcb->hMyObjects[0]);
  };
  return (0);
}
```

Тогда функция **main** будет выглядеть так:

```
int main()
{
  HANDLE hMutexes[n];
  THREADCONTROLBLOCK tcb[n];
  int iThreadID;

  for (int i = 0; i < n; i++)
    hMutexes[i] = CreateMutex(NULL, FALSE, NULL);

  for (int i = 0; i < n; i++)
  {
    tcb[i].iID = i+1;
    tcb[i].hMyObjects[0] = hMutexes[i % n];
    tcb[i].hMyObjects[1] = hMutexes[(i+1) % n];
    CloseHandle(CreateThread(NULL,0,
      (LPTHREAD_START_ROUTINE)ThreadRoutine,
      (void *)&tcb[i],0,LPDWORD(&iThreadID)));
   }
   while(TRUE);
   return(0);
}
```

2.5.3. Анализ параллельной реализации 1

Запустим программу на выполнение несколько раз.

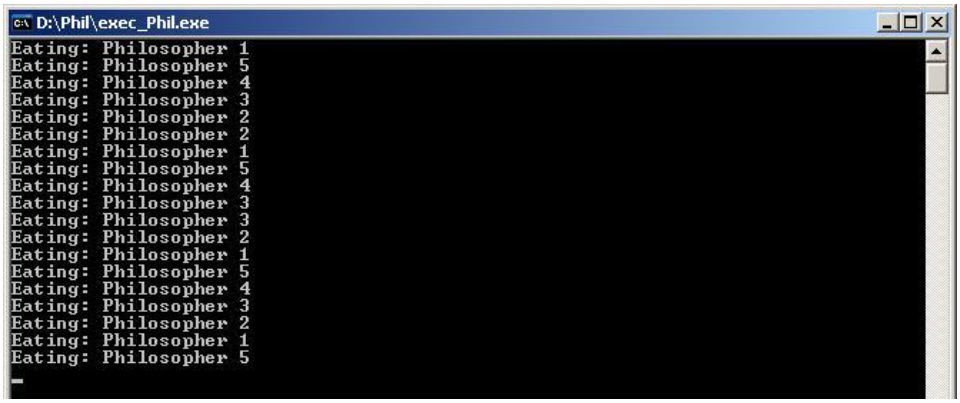

Рис. 15. Задача об обедающих философах. Результаты запуска параллельной реализации 1

Результаты будут варьироваться от запуска к запуску. Единственное, что их объединяет – неизменное зависание в некоторый момент. Попробуем разобраться, в чем дело. Прибегнем к помощи Intel Thread Checker. Как видно из рисунка 16, ITC сгенерировал 5 диагностических сообщений, представляющих для нас интерес. Каждое из сообщений соответствует одному из созданных нами потоков и говорит о наличии тупика.

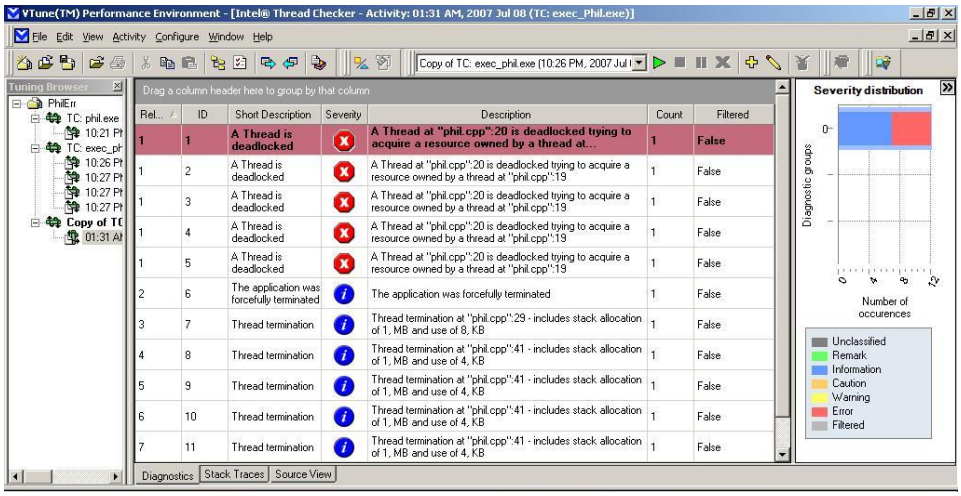

Рис. 16. Диагностика ITC в задаче об обедающих философах (параллельная реализация 1)

2.5.4. Параллельная реализация 2

Подумаем над тем, как исключить тупики, наличие которых обусловливает не столько некорректная реализация, сколько сама постановка задачи. Действительно, возможна ситуация, в которой каждый из философов взял ровно одну вилку и ждет, когда освободится вторая, которая занята соседом. Сосед, в свою очередь, ждет свою вторую вилку и т.д.

Одним из возможных способов решения проблемы является изменение модели поведения философа. К примеру, можно наделить его обязанностью брать вилки одновременно, лишь тогда, когда обе они свободны. Изменения в программной реализации будут минимальны – достаточно заменить 2 вызова функции **`WaitForSingleObject`** на 1 вызов функции **`WaitForMultipleObjects`** для одновременного захвата мьютексов, соответствующих обеим вилкам.

```
#include <stdio.h>
#include <windows.h>

// Количество философов
const unsigned int n = 5;

typedef struct {
  int iID;
  HANDLE hMyObjects[2];
} THREADCONTROLBLOCK, *PTHREADCONTROLBLOCK;

long WINAPI ThreadRoutine(long lParam) {
  PTHREADCONTROLBLOCK pcb=(PTHREADCONTROLBLOCK)lParam;
  while (TRUE) {
    WaitForMultipleObjects(2, pcb->hMyObjects, TRUE, INFINITE);
    printf("Eating: Philosopher %d \n",pcb->iID);
    ReleaseMutex(pcb->hMyObjects[1]);
    ReleaseMutex(pcb->hMyObjects[0]);
  };
return (0);
```

```
}
int main() {
   HANDLE hMutexes[n];
   THREADCONTROLBLOCK tcb[n];
   int iThreadID;

   for (int i = 0; i < n; i++)
     hMutexes[i] = CreateMutex(NULL,FALSE,NULL);

   for (int i = 0; i < n; i++) {
     tcb[i].iID = i+1;
     tcb[i].hMyObjects[0] = hMutexes[i % n];
     tcb[i].hMyObjects[1] = hMutexes[(i+1) % n];
     CloseHandle(CreateThread(NULL,0,
       (LPTHREAD_START_ROUTINE)ThreadRoutine,
       (void *)&tcb[i],0,LPDWORD(&iThreadID)));
   }
   while(TRUE);
   return(0);
}
```

2.5.5. Анализ параллельной реализации 2

Тестовые запуски подтверждают наши ожидания – программа перестала зависать. ITC также не делает по представленному выше коду никаких замечаний.

2.6. Задача о роботе[5]

2.6.1. Постановка задачи

Для постановки задачи рассмотрим некоторую гипотетическую лабораторию искусственного интеллекта, в которой выполняются работы по созданию роботов.

Для испытаний лабораторных образцов построен полигон, устроенный следующим образом: в начале полигона находится единственная дверь. Пройдя через нее, робот оказывается перед B дверьми. Выбрав одну из дверей, робот вновь оказывается перед B дверьми, и т.д. L раз. Пройдя через дверь, робот получает премию в размере x, где x – количество монет, лежащих за данной дверью (для разных дверей количество монет может быть разным). Следует определить среднюю премию, получаемую роботом при проходе через данный полигон.

Считать, что выполняются следующие условия: $B \le 20, L \le 10$.

2.6.2. Модель

Объектом исследования в данной задаче является полигон с иерархической структурой, по которому по определенному алгоритму перемещается робот. Построим математическую модель исследуемого объекта.

Прежде всего, введем необходимые обозначения:

- B – количество вариантов пути, выбираемых роботом на каждом шаге.
- L – количество уровней полигона.

[5] Данная задача при выполнении лабораторной работы рассматривается как дополнительное задание.

- $P_{L,L+1}^{i,j}$ – вероятность попадания из узла *i* уровня *L* в узел *j* уровня *L*+1.
- x_L^i – количество монет, лежащих за дверью *i* уровня *L*.

B, *L*, $P_{L,L+1}^{i,j}$ и x_L^i являются исходными данными задачи. Из условия задачи вытекают следующие ограничения на исходные данные:

- $P_{L,L+1}^{i,j} = 0$, если двери *i* и *j* не соединены.
- $\sum_{j=0}^{B-1} P_{L,L+1}^{i,j} = 1$.

Теперь перейдем непосредственно к выбору модели. Вследствие иерархической структуры полигона выглядит разумным его представление в виде дерева степени *B* и глубины *L*. При этом узлы дерева содержат значения x_L^i, а дугам приписаны вероятности перехода $P_{L,L+1}^{i,j}$.

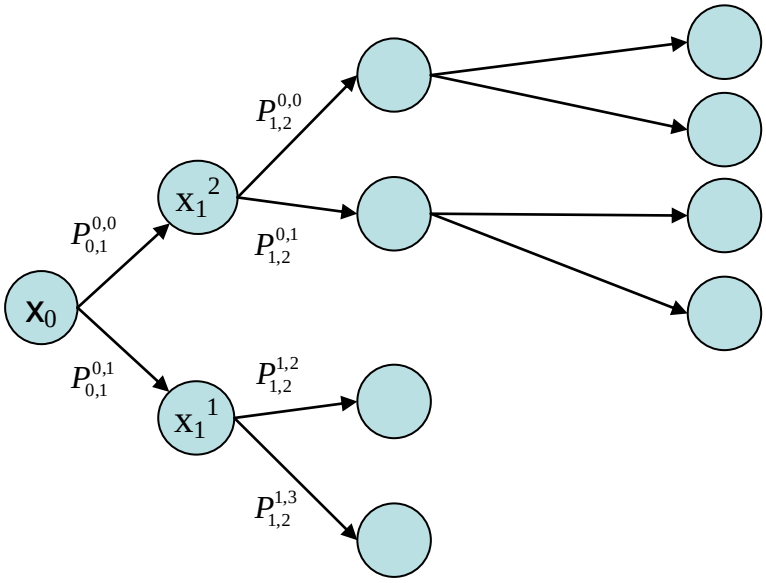

Рис. 17. Модель полигона в задаче о роботе

2.6.3. Метод решения

Перейдем к описанию метода нахождения результата. Пусть E_d^i – средняя премия, которая может быть получена, если робот начинает путь в узле *i* уровня *d*.

Тогда

$$E_d^i = x_d^i + \sum_{j=0}^{B-1} P_{d,d+1}^{i,j} * E_{d+1}^j, \text{ где } d = \overline{0; L-2};\ i = \overline{0; B-1};$$

$$E_{L-1}^{i} = x_{L-1}^{i}, \text{ где } i = \overline{0; B-1}.$$

Очевидный метод вычисления результата E_0^0 состоит в вычислении значений E на последнем уровне дерева (второе соотношение) с последующим пересчетом из конца в начало (первое рекуррентное соотношение).

2.6.4. Последовательная реализация

Перейдем к реализации последовательной версии изложенного выше метода решения.

2.6.4.1. Проектирование структур данных

Предусмотрим для представления дерева константы, хранящие степень B и глубину L. Для организации дерева создадим структуру, соответствующую узлу дерева. Будем хранить в этой структуре количество монет x и вероятность p попадания в узел из узла предыдущего уровня. Учитывая метод решения, состоящий в последовательном пересчете рекуррентных соотношений из пункта 2.6.3, добавим в рассматриваемую структуру специальное поле для хранения текущего значения средней премии E. В результате получим следующие объявления:

```
const int b = 10;     // Степень дерева - B в постановке задачи
const int l = 7;      // Количество уровней дерева

// Максимальное значение в вершине дерева
const int MAX_VALUE = 20;

int NodesCount;       // Количество узлов дерева = (1-b^l) / (1-b)
int BranchesCount;    // Количество ветвей дерева = NodesCount - 1;

typedef struct        // Узел дерева
{
  int value;          // Значение в узле
  // Вероятность попадания в узел из узла предыдущего уровня
  double probability;
  // Компонент для подсчета среднего
  double expectation;
} TreePart;
```

Учитывая регулярную структуру дерева, будем хранить его в виде массива узлов, располагая в нем узлы последовательно по уровням, от корня к листьям. Учитывая возможный большой размер массива, будем создавать его динамически в куче. В результате получим:

```
TreePart *RobotTree;  // Массив для хранения дерева.
// Схема хранения:
// (V00)
// (V10 V11 ... V1b)
// (V21 V22... V2b^2)
// ...
// (Vl-1,1 Vl-1,2...Vl-1,b^(l-1)),
// где скобки стоят для наглядности.
// Дерево упаковывается в одномерный массив по уровням.
```

Для удобства индексации и снижения накладных расходов предусмотрим однократное вычисление значения b в степени от 0 до $l-1$, а также номеров первых узлов каждого уровня дерева в массиве RobotTree.

```
// Вспомогательный массив для хранения значений b
// в степени 0...l-1
__int64 power[l];
// Вспомогательный массив для хранения номера первого узла
// каждого уровня в массиве RobotTree
__int64 index[l];
```

2.6.4.2. Проектирование модульной структуры

Предусмотрим наличие в проекте файла **robot.cpp**, содержащего рассмотренные выше объявления, а также следующих функций:

```
// Ввод дерева
void InputTree(void);
// Освобождение памяти, выделенной для хранения дерева
void ReleaseTree(void);
// Функция вычисления максимума
__inline double max(double v1, double v2) ;
// Функция вычисления премии по значению в узле
__inline double func(double value);
// Вычисление средней премии – основная расчетная функция
double GetExpectation(void) ;
// Головная функция
int main(void) ;
```

Прокомментируем основные функции.

Функция **`InputTree`** предназначена для ввода исходных данных в соответствии с условием задачи. В этой функции вычисляется количество узлов и ветвей, выделяется память для хранения дерева, заполняются вспомогательные массивы **`power`** и **`index`**, инициализируется датчик случайных чисел и происходит заполнение дерева. При этом гарантируется выполнение условия $\sum_{j=0}^{B-1} P_{L,L+1}^{i,j} = 1$. Рекомендуем на этапе отладки отключать инициализацию датчика, для того чтобы при каждом запуске работа велась с одним и тем же деревом, или инициализировать дерево заранее заготовленными данными, для которых вручную подсчитан правильный ответ.

Функция **`func`** предназначена для обобщения задачи на случай, когда размер премии при открывании двери является некоторой функцией от x. В рассматриваемой сейчас постановке она просто возвращает значение x.

Функция **`GetExpectation`** рассчитывает средний размер премии по дереву при помощи описанного выше метода. Приведем ее реализацию:

```
// Вычисление среднего
// Алгоритм базируется на следующих соотношениях:
// E_l,i = СУММА по j=0__b-1
//   (Probability_l+1,j * E_l+1,j + value_l,i),
// где l – уровень, i – номер узла в уровне,
//   l+1,j – узлы потомки узла l,i.
// Учитывая, что на последнем уровне E_l-1,i = Value_l-1,i
```

```
// Вычисляем рекуррентное соотношение из конца в начало.
// В итоге имеем сумму для нулевого узла – корня дерева
double GetExpectation(void)
{
  // Последний уровень
  int i, j, level;
  double sum;
  TreePart rp;
  for (j = 0; j < power[l-1]; j++)
    RobotTree[j + index[l-1]].expectation =
                          func(RobotTree[j + index[l-1]].value);
  // Пересчет из конца в начало
  // Цикл по уровням
  for (level = l - 2; level >= 0; level--)
  {
    // Цикл по узлам уровня
    for (j = 0; j < power[level]; j++)
    {
      // Для узла level, j подсчитываем expectation
      // Цикл по потомкам
      sum  = func(RobotTree[ index[level] + j ].value);
      for (i = 0; i < b; i++)
      {
        rp = RobotTree[ index[level+1] + b * j + i ];
        sum = sum + rp.expectation * rp.probability;
      }
      RobotTree[ index[level] + j ].expectation = sum;
    }
  }
  return RobotTree[0].expectation;
}
```

2.6.5. Параллельная реализация

Рассмотрим возможный вариант распараллеливания предложенной выше последовательной реализации. Акцент сделаем на корректности реализации, а не на ее производительности, которая не является целью данной лабораторной работы. Приведем лишь одно соображение по поводу производительности. Поскольку метод решения задачи предполагает, что последний уровень дерева обсчитывается отдельно от остальных, начнем распараллеливание именно с него. Если подумать, можно обнаружить первый «подводный камень» этой задачи, связанный не с корректностью, но с производительностью разрабатываемой реализации. Легко допустить неточность, посчитав, что распараллеливание на последнем уровне можно опустить. Казалось бы, в чем смысл отдельной работы ради одного уровня? На самом деле смысл есть, поскольку последний уровень содержит наибольшее число узлов и пренебрегать им при распараллеливании не стоит. Применим для распараллеливания обсчета директиву компилятора **`#pragma omp parallel for`**.

Соответствующий фрагмент кода будет выглядеть так:

```
  #pragma omp parallel for
    for (j = 0; j < power[l-1]; j++)
      RobotTree[j + index[l-1]].expectation =
                          func(RobotTree[j + index[l-1]].value);
```

Заметим, что переменная **j** станет локализованной автоматически (согласно стандарту OpenMP), а все остальные по смыслу должны быть общими, поэтому задания дополнительных параметров директивы не требуется.

Перейдем к распараллеливанию основного блока кода, производящего обсчет дерева.

Естественный вариант состоит в разделении всех узлов каждого уровня между потоками. Этого можно добиться по крайней мере двумя способами. Первый состоит в размещении директивы **#pragma omp parallel for** перед циклом по узлам очередного уровня. В итоге получим:

```
    // Пересчет из конца в начало
    // Цикл по уровням
    for (level = l - 2; level >= 0;level--)
    {
    // Цикл по узлам уровня
    #pragma omp parallel for private(sum, i, rp)
      for (j = 0; j < power[level]; j++)
      {
        // Для узла level,j подсчитываем expectation
        sum  = func(RobotTree[ index[level] + j ].value);
        // Цикл по потомкам
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
      }
```

Проблема этого варианта состоит в том, что на каждой итерации внешнего цикла происходит «пробуждение» потоков в начале и «засыпание» в конце, что может плохо отразиться на производительности. Поэтому более правильным является вариант, в котором создание параллельной секции происходит один раз перед внешним циклом.

```
double GetExpectation(void)
{
  int i, j, level;
  double sum;
  TreePart rp;
  // Последний уровень
  #pragma omp parallel for
    for (j = 0; j < power[l-1]; j++)
      RobotTree[j + index[l-1]].expectation =
                              func(RobotTree[j + index[l-1]].value);

  #pragma omp parallel
  {
    // Пересчет из конца в начало
    // Цикл по уровням
    for (level = l - 2; level >= 0; level--)
    {
    // Цикл по узлам уровня
    #pragma omp for private(sum, i, rp)
      for (j = 0; j < power[level]; j++)
```

```
      {
        // Для узла level,j подсчитываем expectation
        sum  = func(RobotTree[ index[level] + j ].value);
        // Цикл по потомкам
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
      }
    }
  }
  return RobotTree[0].expectation;
}
```

2.6.6. Анализ параллельной реализации

Собрав проект в соответствии с рекомендациями, изложенными в описании ITC, запускаем инструмент отладки и обнаруживаем следующие диагностики.

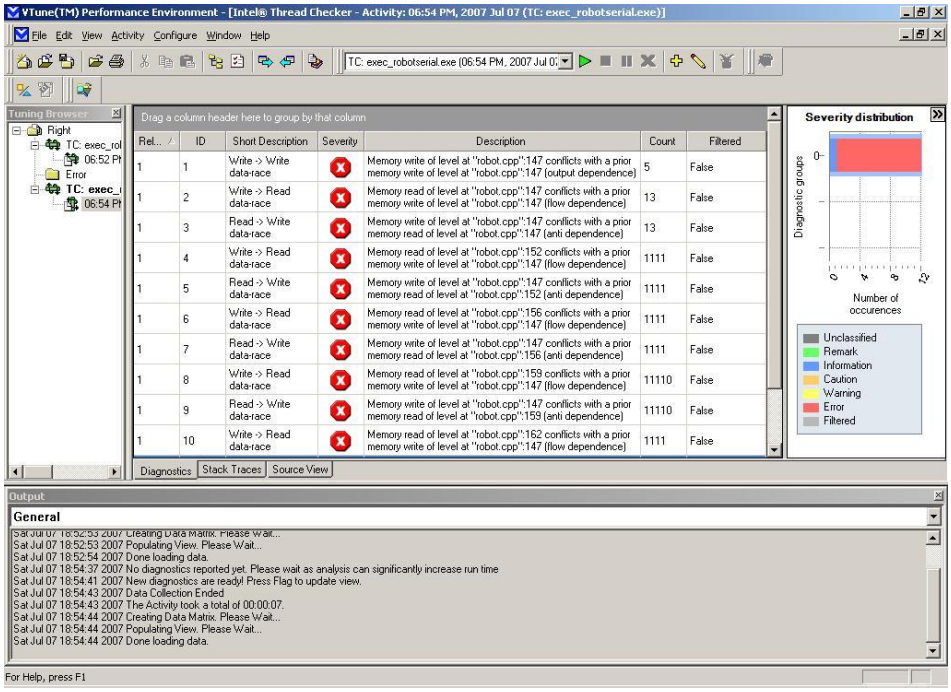

Рис. 18. Диагностика ITC в задаче о роботе

Развернув сообщения об ошибках, мы видим источник проблемы – гонки данных для переменной **`level`**. Действительно, предусмотрев локализацию при распараллеливании цикла, мы забыли об этом в директиве **`#pragma omp parallel`**. Исправим ошибку и приведем в заключение корректную реализацию.

```
double GetExpectation(void)
{
  int i, j, level;
  double sum;
  TreePart rp;
  // Последний уровень
  #pragma omp parallel for
    for (j = 0; j < power[l-1]; j++)
      RobotTree[j + index[l-1]].expectation =
                          func(RobotTree[j + index[l-1]].value);

  #pragma omp parallel private(level)
  {
    // Пересчет из конца в начало
    // Цикл по уровням
    for (level = l - 2; level >= 0; level--)
    {
    // Цикл по узлам уровня
    #pragma omp for private(sum, i, rp)
      for (j = 0; j < power[level]; j++)
      {
        // Для узла level,j подсчитываем expectation
        sum  = func(RobotTree[ index[level] + j ].value);
        // Цикл по потомкам
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
      }
    }
  }
  return RobotTree[0].expectation;
}
```

2.7. Дополнительные задания

1. Изучите стандартные примеры, поставляемые вместе с инструментом отладки Intel Thread Checker.

2. Изучите постановку задачи умножения матрицы на вектор, последовательные реализации различных алгоритмов, а также возможные пути распараллеливания (см., например, раздел «Параллельные методы умножения матрицы на вектор» курса «Введение в методы параллельного программирования» – http://www.software.unn.ru/ccam/multicore/programm.html). Проанализируйте прилагаемые параллельные реализации, содержащие ошибки (папка Code\MV). Выполните отладку прилагаемых программ, добейтесь их работоспособности.

3. Изучите постановку задачи умножения матрицы на матрицу, последовательные реализации различных алгоритмов, а также предлагаемые пути распараллеливания (см., например, раздел «Параллельные методы матричного умножения» курса «Введение в методы параллельного программирования» – http://www.software.unn.ru/ccam/multicore/programm.html). Проанализируйте прилагае-

мые параллельные реализации, содержащие ошибки (папка Code\MM). Выполните отладку прилагаемых программ, добейтесь их работоспособности.

4. Подумайте над задачей об обедающих философах. Рассмотрите другие варианты ее решения. Реализуйте их. Выполните отладку разработанных программ, добейтесь их работоспособности. В качестве одного из средств контроля используйте ITC.

2.8. Литература

Использованные источники

1. Intel® Thread Checker for Windows*. Getting Started Guide. Version 3.0. — Intel Corporation, 2006.
2. Intel® Thread Checker Help. Version 3.0. — Intel Corporation, 2006.
3. Intel® Thread Checker. Guide to Sample Code. Version 3.0. — Intel Corporation, 2006.

Рекомендуемая литература

4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003 (Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley, 2000).
5. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004.
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
7. Гергель В.П. Теория и практика параллельных вычислений. М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
8. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.

Дополнительная литература

9. Березин И.С., Жидков И.П. Методы вычислений. М.: Наука, 1966.

Информационные ресурсы сети Интернет

10. Сайт Лаборатории параллельных информационных технологий НИВЦ МГУ – http://www.parallel.ru.
11. Официальный сайт OpenMP – www.openmp.org.
12. Официальный форум MPI – www.mpi-forum.org.

Часть II. Оптимизация параллельной программы

3. Intel Thread Profiler. Краткое описание

3.1. Введение

Многопоточный подход к разработке приложений появился в программировании уже достаточно давно и широко используется для повышения производительности. С появлением многоядерных процессоров интерес к многопоточному программированию значительно усилился. Многоядерные системы прошли путь от лабораторных образцов до настольных компьютеров, доступных рядовым пользователям. В 2005 году началась массовая продажа процессоров с двумя ядрами, в 2007 – четырехъядерных процессоров. В будущем ожидается сохранение тенденции увеличения числа ядер, и можно с уверенностью сказать, что с течением времени систем, построенных не на базе многоядерных процессоров не останется.

В подобной ситуации остро встает вопрос об эффективном использовании многоядерных архитектур. Многопоточное программирование представляет собой очень удобный подход, позволяющий создавать приложения, способные рационально использовать процессоры с несколькими ядрами, обеспечивая их оптимальную загрузку. Уже сегодня подавляющее большинство существующих приложений использует несколько потоков: браузеры, файловые менеджеры, словари, антивирусы и многие другие. Можно ожидать, что в дальнейшем все больше приложений будут изначально разрабатываться как многопоточные.

Необходимо, тем не менее, отметить, что разработка многопоточных приложений зачастую на порядок сложнее, чем последовательных. Новая парадигма приносит не только новые возможности, но и новые сложности. Создание эффективных многопоточных приложений – весьма трудоемкий процесс, а значит, имеется насущная необходимость в инструментальной поддержке всех стадий разработки подобных приложений, начиная с построения архитектуры и заканчивая настройкой параметров и финальной оптимизацией.

В настоящее время существует достаточно много инструментов, ориентированных на поддержку многопоточного программирования. Компания Intel выделяется на общем фоне тем, что предлагает целое семейство подобных инструментов, представителем которого является Intel® Thread Profiler (ITP).

3.1.1. Семейство инструментов Intel для поддержки разработки многопоточных приложений

Цикл разработки программных приложений включает в себя следующие основные стадии: анализ требований, разработка архитектуры, реализация, тестирование, отладка и оптимизация. Зачастую при этом приходится проходить каждую стадию несколько раз. Эта деятельность разработчиков требует наличия специализированных инструментов, и в настоящее время компания Intel предлагает целый ряд программных продуктов, предназначенных для поддержки разработки многопоточных приложений:

- Intel® C/C++ Compiler и Intel® Fortran Compiler – компиляторы;
- Intel® Performance Libraries – библиотеки математических функций (численные методы, обработка сигналов, генераторы случайных чисел);
- Intel® VTune Performance Analyzer – профилировщик;

- Intel® Thread Checker – отладчик многопоточных приложений;
- Intel® Thread Profiler – профилировщик многопоточных приложений;
- Intel® Threading Building Blocks – C++-библиотека времени выполнения, представляющая набор примитивов для разработки многопоточных приложений (параллельные алгоритмы, потокобезопасные структуры данных, примитивы синхронизации).

Данное семейство инструментов предназначено для поддержки всего цикла разработки многопоточных приложений. Их конечная цель – значительно ускорить процесс разработки, обеспечив разработчиков удобными и эффективными инструментами. Инструменты поддерживают различные методы создания многопоточных приложений: OpenMP, Pthreads, Win32 Threading API, автоматическое распараллеливание. Более подробную информацию можно найти на сайте http://www.intel.com/software/products.

На рис. 1 схематически представлен цикл разработки многопоточного приложения с указанием программных продуктов компании Intel, предназначенных для помощи на соответствующих стадиях.

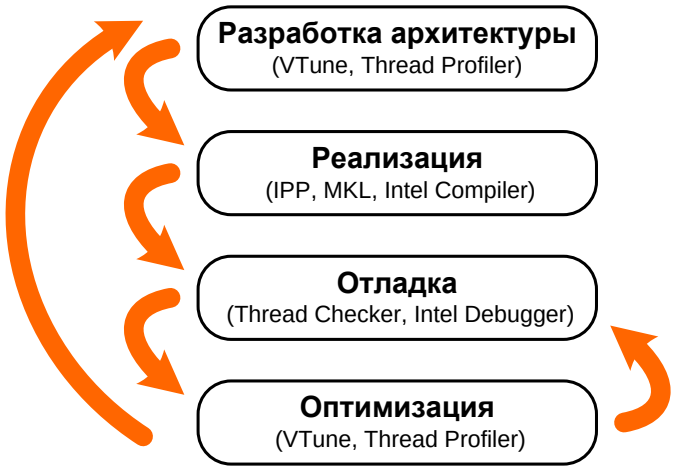

Рис. 1. Цикл разработки приложения

Как видно из диаграммы, ITP в основном призван помочь разработчикам на финальных стадиях создания многопоточного приложения. Он предназначен для профилирования многопоточных приложений с целью их последующей оптимизации и финальной настройки. Однако следует заметить, что ITP может оказаться полезным еще на стадии разработки архитектуры. Разработчик имеет возможность сравнивать между собой различные архитектурные решения, создавая прототипы и сравнивая их производительность при помощи ITP и VTune.

3.1.2. Определение и цели профилирования

Центральная идея ускорения последовательных приложений достаточно проста. Необходимо выявить участки кода, в которых приложение проводит основную часть времени, и оптимизировать их. Известен эвристический закон «20/80», утверждающий, что приложение проводит 80% времени работы в 20% кода. Таким образом, за счет оптимизации именно этих 20% кода можно добиться значитель-

ного повышения производительности. Как раз для поиска таких узких мест используются специальные средства, называемые профилировщиками.

Под понятием *профилирование* обычно понимается процесс анализа производительности приложения и учета потребляемых им ресурсов. Тем самым, *профилировщик* – это инструмент, при помощи которого осуществляется профилирование.

Профилировщики исследуют поведение программы в процессе ее выполнения. В частности, собирается такая информация, как частота вызовов функций, продолжительность их работы, число потоков, время ожидания, – любая информация, способная помочь при оптимизации производительности. В числе приемов, используемых профилировщиками для сбора информации, инструментация кода, аппаратные прерывания, внедрение счетчиков производительности и многие другие. Выходной информацией профилировщика является *трасса* – список событий, произошедших в приложении (исполненные инструкции, вызовы функций, вызовы операционной системы). После получения трассы профилировщик обрабатывает собранную информацию и строит *профиль* приложения – статистическую сводку о работе приложения. Затем профиль представляется в графическом виде, удобном для анализа. Разработчик изучает его, определяет проблемные места и ищет способы оптимизации своего приложения.

Ситуация с многопоточными приложениями существенно другая, поскольку в силу вступают новые аспекты производительности, связанные с взаимодействием потоков, затратами на их создание и синхронизацию. Ускорение отдельных участков кода может не дать никакого прироста производительности, если, например, разработчик построил свое приложение таким образом, что основную часть времени потоки ожидают освобождения разделяемого ресурса. Также можно упомянуть такие распространенные в многопоточных приложениях проблемы, как неравномерное распределение нагрузки и неэффективное использование примитивов синхронизации. Эти факторы могут привести к катастрофически низкой производительности приложения, вплоть до того, что многопоточная версия будет медленнее последовательной.

Таким образом, для анализа новых аспектов программной оптимизации, присущих именно многопоточным приложениям, необходимы специальные инструменты, ориентированные на обнаружение проблемных ситуаций и последующую помощь программистам в их разрешении. Одним из самых эффективных и популярных представителей инструментов такого класса является ITP.

3.1.3. Назначение Intel Thread Profiler

ITP предназначен для профилирования многопоточных приложений и помощи разработчику в процессе оптимизации программного кода. Следует напомнить, что ITP также может быть использован на стадии разработки архитектуры приложения. Разработчики могут создать несколько прототипов, соответствующих различным архитектурным решениям, и сравнить их производительность при помощи ITP. Также ITP может быть использован для анализа масштабируемости приложения. Например, если заранее известны характеристики компьютерной системы (число процессоров или ядер), на которой будет функционировать приложение, то ITP можно использовать для подбора оптимального числа потоков.

Итак, ITP может оказать разработчикам существенную помощь при решении проблем, связанных с производительностью многопоточного приложения. Перечислим основные варианты использования Intel® Thread Profiler:

- анализ эффективности распределения вычислительной нагрузки между потоками (лабораторная работа 1, раздел 4);
- анализ эффективности обращения к разделяемым ресурсам (лабораторная работа 2, раздел 5);
- определение наиболее медленных потоков и, как результат, нуждающихся в оптимизации (лабораторная работа 1);
- выбор оптимальной архитектуры многопоточного приложения (число потоков, алгоритм, примитивы синхронизации) (лабораторная работа 2);
- анализ эффективности работы с потоками и примитивами синхронизации (лабораторная работа 2);
- анализ масштабируемости приложения на вычислительных узлах с различным числом процессоров.

Таким образом, ITP является весьма удобным и эффективным инструментом для поиска причин низкой производительности многопоточного приложения и последующей помощи в их устранении.

3.2. Технические характеристики Intel Thread Profiler

3.2.1. Поддерживаемые методы создания многопоточных приложений

ITP позволяет профилировать многопоточные приложения, созданные на основе технологий OpenMP, Windows threads, POSIX threads.

Инструмент имеет два режима: OpenMP и Threaded. Первый из них используется для анализа приложений, созданных с использованием OpenMP и скомпилированных компиляторами компании Intel. Этот режим предлагает возможности сбора и отображения следующей информации:

- затраты на синхронизацию, накладные расходы на поддержку работы с потоками;
- дисбаланс вычислительной нагрузки;
- сравнение результатов разных запусков.

Главной особенностью этого режима является возможность оценивания потенциальной масштабируемости, что очень удобно при создании параллельного прототипа приложения.

Второй режим является более общим – он обеспечивает анализ OpenMP-приложений и многопоточных приложений, основанных на использовании Windows threads или POSIX threads. В этом режиме отображается следующая информация:

- аналогичные OpenMP-режиму данные по потокам программы и уровню параллелизма;
- критический путь программы;
- распределение временных затрат по критическому пути на исполнение, синхронизацию, ожидание, блокировку;

- связь потоковых событий с исходным кодом.

Режим Threaded предоставляет более богатые возможности, поэтому мы сосредоточимся на рассмотрении именно этого режима.

3.2.2. Системные требования

Аппаратное обеспечение

Минимальные требования:

- Процессор Intel® Pentium 4;
- ОЗУ 512 Мб;
- Свободное дисковое пространство 300 Мб.

При практическом использовании ITP для достижения приемлемых показателей оперативности работы могут быть рекомендованы повышенные требования к минимально необходимым аппаратным ресурсам:

- Процессор Intel® Pentium 4, поддерживающий технологию Hyper-Threading, или процессор Intel® Xeon;
- ОЗУ 2 Гб.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

Программное обеспечение

Минимальные требования:

- Microsoft® Windows® XP Professional, или Microsoft® Windows® Server 2003;
- Microsoft® Visual C++® 6.0 или выше;
- Microsoft® Internet Explorer® 6.0 или выше;
- Adobe® Acrobat® Reader.

Для анализа OpenMP-приложений и инструментации на уровне исходных кодов требуется следующее программное обеспечение:

- Intel® C++ Compiler 8.1 или выше для Windows® для архитектуры IA-32;
- Intel® C++ Compiler 9.1 или выше для Windows® для архитектуры Intel® EM64T;
- Intel® Fortran Compiler для Windows® 8.1, Package ID: w_fc_pc_8.1.023 или выше.

3.3. Основные концепции и понятия профилирования

3.3.1. Понятие критического пути

Первое понятие, которое нам понадобится, – *критический путь* (*critical path*). Оно является ключом к пониманию всего процесса профилирования, поэтому очень важно хорошо усвоить его. Прежде чем дать формальное определение этого понятия, попытаемся понять его суть на примере.

Представим себе следующее многопоточное приложение: имеется основной поток, который подготавливает данные и производит их первичную обработку, а

также один дочерний поток, который завершает обработку и выводит результаты. Простейшая иллюстрация этого примера – физическое моделирование, в котором основной поток занят вычислениями (например, решением разностной схемы), а дочерний – подготовкой к визуализации и непосредственно визуализацией результатов вычислений. Часто подобные взаимодействия реализуются в виде конвейера, поскольку в то время как один из потоков работает с устройствами ввода-вывода, второй поток может нагружать процессор вычислениями.

В нашем случае, как только первый поток вычислил первую порцию данных для отображения, он может начинать вычисление следующей порции, а второй поток параллельно с этим начнет визуализацию. Важно при этом обеспечить синхронизацию между потоками, чтобы не допустить уничтожения данных, до того как они были визуализированы.

Ниже (рис. 2) графически представлено взаимодействие потоков в случае, если бы требовалось выполнить только две итерации. Потокам в нашем приложении соответствуют горизонтальные линии, сплошные участки – активному состоянию потоков, а пунктирные – состоянию ожидания. Диагональные линии соответствуют сигналам, передаваемым между потоками, а проекции этих линий на ось времени – временам прохождения сигналов. Кружками обозначены все события, в результате которых изменяются состояния потоков (создание/завершение, блокировка/активизация).

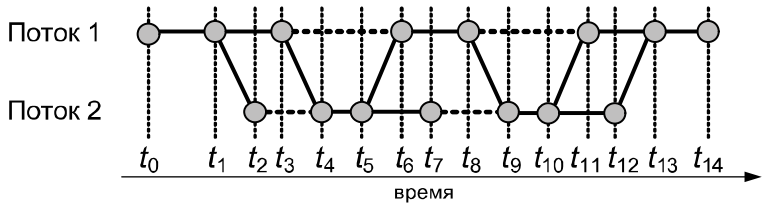

Рис. 2. Пример критического пути многопоточного приложения

Рассмотрим диаграмму подробнее. В начальный момент времени существовал только один поток, выполняющийся последовательно. Затем в момент времени t_1 происходит создание дочернего потока, который начинает функционировать в момент времени t_2, но сразу попадает в состояние ожидания, поскольку данные для визуализации еще не подготовлены. Затем в точке t_3 завершаются вычисления, и дочерний поток начинает визуализацию подготовленных данных, которая продолжается с момента t_4 до момента t_7. Заметим, что второй поток может освободить массивы с результатами вычислений сразу после того, как переведет их в формат, необходимый для графического представления (момент времени t_5). За счет этого и достигается распараллеливание в рассматриваемом приложении. Можно видеть, что потоки работают параллельно на промежутке времени между моментами t_6 и t_7. Далее осуществляется еще одна итерация, после чего в момент времени t_{12} второй поток завершает свое исполнение, а в момент времени t_{14} завершается весь процесс.

Рассмотренная диаграмма представляет собой граф. Вершинам, как уже говорилось, соответствуют события, изменяющие состояние потоков, а ребрам – действия, выполняемые потоками (мы не учитываем пунктирные линии). При этом ребрам можно сопоставить вес, равный времени, затрачиваемому на выполнение

действия. *Путем приложения* называется любой путь в указанном графе, соединяющий вершины, соответствующие началу и завершению приложения. *Критическим путем приложения* называется путь, имеющий максимальную сумму весов входящих в него ребер. Приложения со многими потоками могут иметь большое число путей, однако в нашем случае приложение имеет всего два пути, каждый из которых можно назвать критическим, так как суммы их весов совпадают.

Важность понятия критического пути определяется тем, что *сумма весов ребер в критическом пути равна времени выполнения приложения*. Из этого факта вытекает следующий принцип: необходимо оптимизировать прежде всего те участки приложений, которые вошли в критический путь, поскольку именно они определяют производительность.

Таким образом, профилирование и оптимизация приложения неразрывно связаны с процессом анализа критического пути. Критический путь – это первое, на что нужно обращать внимание, поскольку не входящие в него участки приложения вносят меньший вклад в суммарное время работы. Далее мы увидим, что ITP позволяет строить критический путь приложения и предоставляет различные средства для его анализа.

3.3.2. Состояния потоков

При анализе критического пути пользуются таким понятием, как *состояние потока* (*thread state*). А именно, поток может находиться в трех состояниях:

- *Активное состояние* (*active*) – поток исполняется в настоящее время;
- *Состояние активного ожидания* (*spin*) – постоянно повторяемая проверка состояния блокировки (например, при использовании `pthread_spin_lock()`);
- *Состояние ожидания* (*wait*) – ожидание завершения какой-либо блокирующей операции (например, операции ввода-вывода).

3.3.3. Понятие категорий времени

Рассмотрим теперь характеристики эффективности использования аппаратных ресурсов активными потоками приложения. Для этого в ITP используются два параметра: *уровень параллелизма* (*concurrency*) и *поведение* (*behavior*).

Уровни параллелизма используются для обозначения того, насколько полно потоки нагружают процессоры вычислительного узла. Так, одновременная работа двух потоков на двухъядерном процессоре означает эффективное использование его ресурсов (*full utilization*), на одноядерном – сверхиспользование (*over utilization*), а на четырехъядерном – недостаточно эффективное использование (*under utilization*).

Существует несколько типов поведения потока в активном состоянии:

- *Сдерживание* (*impact*) – ситуация, когда поток сдерживает выполнение другого потока (например, поток занял мьютекс, освобождения которого ожидают другие потоки);
- *Блокировка* (*blocking*) – приостановка потока в результате вызова блокирующих функций (например, операций ввода/вывода);

- *Критический путь* (*critical path*) – ситуация, когда поток выполняет участок кода, входящий в критический путь, при отсутствии других ожидающих потоков.

Уровень параллелизма и поведение независимы друг от друга, и в совокупности они называются *категориями времени* (*time categories*) и характеризуют эффективность работы активных потоков. Собственно весь анализ производительности состоит в том, чтобы оценить каждый участок критического пути по этим двум параметрам и определить наиболее проблемные участки приложения.

В ITP используется несколько видов представления критического пути, каждое из которых мы впоследствии подробно изучим. Для удобства анализа каждый участок критического пути окрашивается в определенный цвет в зависимости от уровня параллелизма и поведения потока на этом участке. На рис. 3 представлена сводная таблица используемых цветов; *n* обозначает число активных потоков приложения, а *p* – число процессоров (ядер).

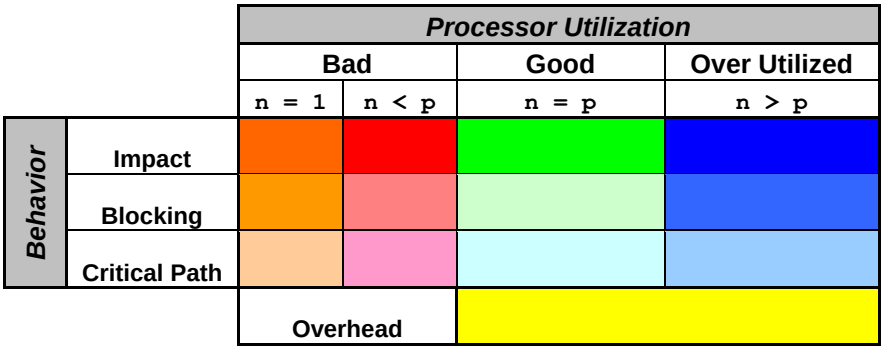

Рис. 3. Цвета, используемые для обозначения категорий времени

Рассмотрим приведенную таблицу подробнее, чтобы знать впоследствии, на какие цвета нужно обращать внимание прежде всего. Сначала исследуем связь числа потоков и числа ядер. Как видно из таблицы, для обозначения этого соотношения используется четыре цвета: оранжевый, красный, зеленый и синий. Зеленый цвет – признак оптимального соотношения, когда число ядер и потоков совпадает. Синий цвет используется для обозначения ситуаций, когда одновременно работает больше потоков, чем доступно ядер. Как результат, потоки могут мешать работе друг друга. Присутствие красного и оранжевого цветов в окраске критического пути чаще всего свидетельствует о необходимости перестройки приложения. Эти цвета означают, что используются не все ядра процессора и можно получить дополнительное ускорение за счет более эффективного распараллеливания.

Рассмотрим теперь второй параметр – тип поведения потоков. Для обозначения типа поведения потока цветом используется изменение яркости. При этом используется простое правило: чем ярче цвет участка, тем большего внимания от разработчика он требует. Самый яркий цвет используется для указания *impact*-состояния потока, поскольку прежде всего необходимо сокращать время ожидания потоков. Далее по важности следует *blocking*-состояние, которое тоже требует пристального контроля, так как нужно минимизировать время ожидания потоков. Последним идет состояние *critical path*.

Имеется особая категория времени, обозначаемая желтым цветом и указывающая на *непроизводительные издержки* (*overhead*) в многопоточном приложении. Под непроизводительными издержками понимаются накладные расходы на создание потоков, управление и синхронизацию между потоками. Другими словами, эта величина показывает количество процессорного времени, израсходованного на поддержку работы потоков. Чаще всего большая доля желтого цвета в критическом пути свидетельствует о чрезмерно частом вызове функций синхронизации и необходимости перепроектирования приложения.

Теперь, зная о категориях времени, мы можем сказать, какие цвета будут содержаться на критическом пути нашего примера из раздела 3.3.1.

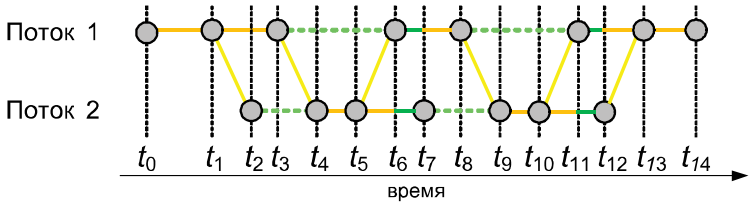

Рис. 4. Пример раскраски критического пути многопоточного приложения в соответствии с категориями времени

Итак, подведем итоги: профилирование многопоточного приложения основано на построении критического пути и его анализе с точки зрения эффективности использования процессора. Для этого выделяют категории времени, часть которых считаются проблемными и свидетельствуют о неправильной организации многопоточного приложения. Деятельность разработчика, таким образом, состоит в нахождении проблемных участков критического пути, выявлении причин (соотнесение с исходным кодом) и в последующем их разрешении.

3.4. Проблемы производительности, определяемые при помощи профилирования

Мы уже говорили ранее, что профилирование многопоточных приложений имеет свою специфику. ITP ориентирован на выявление проблемных ситуаций, характерных именно для многопоточных приложений, таких как неэффективное управление потоками, ошибки при выборе и использовании примитивов синхронизации, неправильное распределение вычислительной нагрузки и другие недостатки.

Рассмотрим далее наиболее распространенные ошибки и то, как ITP может помочь разработчикам в их обнаружении и исправлении.

3.4.1. Распределение вычислительной нагрузки

Весьма распространенной ошибкой является неравномерное распределение нагрузки между потоками. Общее время работы приложения в значительной степени зависит от времени работы самого медленного из его потоков. Очевидно, приложение будет работать быстрее всего, если нагрузка поделена между потоками поровну – тогда они завершают работу одновременно и за минимальное время.

Типичный пример: стоит задача обработки множества заявок, трудоемкость каждой из которых заранее неизвестна. В такой ситуации не всегда правильно раз-

делять множество заявок поровну между потоками. Дело в том, что один поток может, например, получить все сложные заявки, в результате чего он станет узким местом в приложении.

Проблема распределения вычислительной нагрузки в ряде ситуаций решается достаточно просто. Первый признак ее появления – большая доля последовательных вычислений в приложении, что легко обнаруживается при анализе критического пути. Кроме того, ITP позволяет определить время работы каждого из потоков. Если при этом наблюдается существенное различие между временами работы нескольких потоков (при том что они выполняют одинаковые действия), это свидетельствует о неравномерном распределении нагрузки.

Для рассмотренного выше примера в качестве решения можно предложить не делать априорного разделения множества заявок между потоками. Вместо этого стоит разрешить потокам выбирать из общей очереди новую заявку каждый раз после того, как была обработана предыдущая. Тогда пока один поток обрабатывает одну сложную заявку, второй поток успеет обработать несколько простых, и в результате можно ожидать более сбалансированного времени работы потоков.

3.4.2. Синхронизация и производительность

Аспекты производительности, связанные с синхронизацией между потоками, требуют особого внимания. Неправильно выбранная стратегия может привести к тому, что параллельная версия приложения будет выполняться даже медленнее, чем последовательная. Поэтому разработчик должен тщательно продумать используемую в его приложении модель синхронизации, и ITP способен оказать ему в этом существенную помощь.

Рассмотрим основные вопросы, связанные с синхронизацией.

3.4.2.1. Выбор примитивов синхронизации

Существует достаточно большое число примитивов синхронизации: мьютексы, семафоры, мониторы, критические секции. Все они имеют одинаковое назначение (обрамление критических областей программы), но эффективность (дополнительные накладные расходы) их работы может существенно различаться. При этом разработчику важно выбрать наиболее подходящий тип примитива синхронизации для каждой конкретной ситуации.

Рассмотрим, например, ситуацию, когда в приложении имеется некоторый целочисленный счетчик, являющийся глобальной переменной. При этом если мы в приложении используем каждый раз увеличение этого счетчика на единицу, то выбор такого объекта, как мьютекс, является нерациональным. Гораздо эффективнее использовать атомарную функцию ОС Windows **`InterlockedIncrement`**. Кроме того, при разработке модели синхронизации следует делать выбор в пользу примитивов пользовательского уровня (**`CriticalSection`**, например), поскольку они работают быстрее, так как не генерируют системных вызовов операционной системы.

ITP способен помочь в выборе наиболее подходящего примитива, позволяя определить время, затраченное на работу с каждым из объектов синхронизации. Таким образом, разработчик может реализовать несколько моделей синхронизации и сравнить их производительность между собой.

3.4.2.2. Синхронизация между потоками

Разработчикам следует придерживаться следующей рекомендации: производить синхронизацию между потоками как можно реже. То есть необходимо сделать потоки максимально независимыми, чтобы избежать ситуаций, когда они ожидают друг друга. Слишком частое обращение нескольких потоков к разделяемому ресурсу приводит к тому, что большое количество времени потоки простаивают, находясь в состоянии ожидания.

Часто встречающаяся ситуация – это совместное использование одного и того же объекта несколькими потоками. Чтобы избежать блокировки потоков при обращении к объекту, часто используется подход, при котором каждый из потоков получает копию объекта в свое личное пользование. Так, в частности, поступают при работе нескольких пользователей с одной таблицей базы данных.

В данной ситуации ITP используется следующим образом. С его помощью определяются объекты, обращение к которым происходит наиболее часто. Затем анализируется, насколько затратно столь частое обращение к объекту. Если обнаруживается, что производительность существенно страдает, разработчику следует попытаться изменить архитектуру приложения. Хороший пример – замена глобальных переменных локальными.

3.4.3. Непроизводительные издержки при работе с потоками

Важный момент при работе с потоками – появление дополнительных накладных расходов. Конечно, эти затраты гораздо меньше, чем при управлении процессами, но все равно они могут оказаться весьма существенными. Основная рекомендация здесь состоит в следующем: потоки за время своей жизни должны совершать работу гораздо большей сложности, чем трудоемкость их собственного создания, уничтожения и управления ими.

Понятно, что если накладные расходы на управление потоками будут преобладать над их полезной деятельностью, то производительность приложения только пострадает.

ITP позволяет определить долю непроизводительных издержек от общего времени работы приложения. Если она оказывается слишком велика, скорее всего приложение требует перепроектирования. Так, например, если создается система, обслуживающая поступающие в режиме реального времени запросы, то для обработки лучше содержать пул потоков, вместо того чтобы создавать новый поток для каждого очередного запроса.

3.5. Общий порядок работы с инструментом

Порядок работы с Intel® Thread Profiler включает в себя следующие шаги:

- инструментация приложения;
- профилирование приложения;
- анализ полученной информации.

На первом шаге происходит подготовка приложения к профилированию – так называемая инструментация. Затем осуществляется запуск, и в процессе выполнения происходит накопление статистической информации о работе приложения. Собранная информация представляет собой трассу приложения, которая затем

обрабатывается ITP и представляется в графическим виде, удобном для понимания. После этого разработчик может непосредственно начинать анализ производительности приложения.

Далее мы приведем рассмотрение процессов инструментации и профилирования. О том, как эти процессы осуществляются в ITP, мы расскажем в разделе 3.6.3.

3.5.1. Инструментация приложения

Инструментация представляет собой встраивание в приложение дополнительных вызовов, при помощи которых профилировщик получает информацию о работе приложения. Эти вызовы могут быть добавлены как на уровне исходного кода приложения, так и в уже скомпилированное приложение. В связи с этим различают *инструментацию исходных кодов* (*source instrumentation*) и *бинарную инструментацию* (*binary instrumentation*). ITP поддерживает оба этих способа.

Бинарная инструментация выполняется автоматически при запуске приложения из ITP. То есть вы имеете возможность взять любое уже скомпилированное приложение и немедленно начать профилировку. Однако полезность информации, которую вы получите, существенно зависит от опций, с которыми было скомпилировано приложение. Так, если в него была включена отладочная информация, то вы сможете обращаться к исходному коду из ITP. В противном случае вам будет доступен лишь ассемблерный код, что может быть весьма неудобно.

В связи с этим обычная процедура состоит в следующем: разработчик компилирует свое приложение со всеми необходимыми опциями, а затем использует бинарную инструментацию. Мы рассмотрим этот процесс подробнее в разделе 3.6.2.

Инструментация исходных кодов используется крайне редко. Создатели ITP рекомендуют ее лишь в двух ситуациях:

- Бинарная инструментация недоступна. Это справедливо для систем, созданных для работы на архитектурах Intel® Itanium и Intel® EM64T.
- Необходимо запустить инструментированное приложение вне Intel® Thread Profiler.

Далее под инструментацией мы будем понимать именно бинарную инструментацию, поскольку именно с ней нам придется работать.

3.5.2. Профилирование приложения

После того как приложение было инструментировано, можно начинать профилирование. ITP осуществляет запуск приложения, в процессе которого собирает его трассу. В нее включается следующая статистическая информация:

- идентификаторы созданных потоков;
- время создания и уничтожения каждого потока;
- количество времени, которое каждый поток провел в состоянии ожидания;
- эффективность использования приложением ядер процессора на каждом участке критического пути.

ITP также регистрирует большое число других событий и вызовов API, информация о которых может быть полезна при оптимизации производительности многопоточного приложения.

При профилировании старайтесь следовать следующим советам:

- Избегайте запускать другие приложения во время профилирования. Деятельность посторонних приложений (особенно потребляющих много ресурсов) может существенно исказить интересующую вас информацию.
- Производите профилирование несколько раз и для анализа выбирайте тот запуск, когда приложение отработало быстрее всего. Этот запуск соответствует ситуации, когда на ваше приложение было меньше всего воздействий, поэтому эта информация ближе всего к «идеальному» профилю вашего приложения.

Далее мы на конкретном примере познакомимся с графическим интерфейсом ITP и основными приемами работы с ним.

3.6. Пример использования Intel Thread Profiler

Целью настоящего раздела является начальное ознакомление с инструментом ITP и общими принципами работы с ним. Изучается процесс подготовки приложения к профилированию, графический интерфейс ITP и основные возможности по анализу производительности многопоточных приложений.

Настоящий раздел представлен в виде лабораторной работы, которую читатели могут выполнять одновременно с чтением.

3.6.1. Изучение профилируемого приложения

Лабораторная работа проводится на учебном приложении, которое осуществляет факторизацию (разложение на простые множители) чисел из диапазона от 1 до *N*. Используется алгоритм, который основан на попытке деления факторизуемого числа на каждое из чисел меньше его. Если остаток от деления равен нулю, то очередной множитель запоминается, после чего производится повторная попытка деления на это же число. При нахождении каждого множителя факторизуемое число делится на него, и алгоритм завершает работу, когда частное от очередного деления становится равным единице. Заметим, что это малоэффективный алгоритм факторизации, поэтому мы не рекомендуем использовать его при решении практических задач. Такой алгоритм используется только в учебных целях – на его примере мы сможем изучить некоторые особенности оптимизации многопоточных приложений.

Откройте проект **Factorization**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**…;
- в диалоговом окне **Open Project** выберите папку **C:\ITPLab\Factorization**;
- дважды щелкните на файле **Factorization.sln** или, выбрав файл, выполните команду **Open**.

В окне **Solution Explorer** дважды щелкните на файле исходного кода **Factorization.cpp** (рис. 5). После этого в рабочей области **Microsoft Visual Studio** появится программный код, с которым нам предстоит работать.

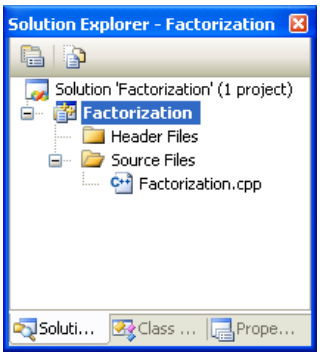

Рис. 5. Открытие файла Factorization.cpp

Приступим к изучению приложения. В начале файла **Factorization.cpp** объявлены две константы:

```
#define NUM_NUMBERS 100000
#define NUM_THREADS 2
```

Первая из них указывает количество чисел, которые будут факторизованы. В данном случае будет построено разложение для чисел от 1 до 100000. Вторая константа показывает, сколько потоков будет создано для решения этой задачи.

Также объявлен глобальный массив векторов **`divisors`**:

```
vector<int> divisors[NUM_NUMBERS+1];
```

Он предназначен для хранения простых множителей каждого из чисел. Так, например, вектор **`divisors`** для `NUM_NUMBERS=6` будет содержать два числа: 2 и 3.

В данной лабораторной работе нас будут интересовать только функции **`main`** и **`factorization1`**.

Ознакомьтесь с кодом функции **`main`**. Он содержит объявления переменных, операции создания потоков и ожидания их завершения, а также вывод на экран, предназначенный для контроля правильности результатов.

После этого ознакомьтесь с функцией **`factorization1`**, которая представляет собой рабочую функцию потока. В ней реализуется простейшая стратегия распределения нагрузки между потоками. А именно, первый поток строит разложение для первых **`NUM_NUMBERS/NUM_THREADS`** чисел, второй – для второго массива чисел такой же длины, и так далее. Пример распределения нагрузки между двумя потоками представлен на рис. 6.

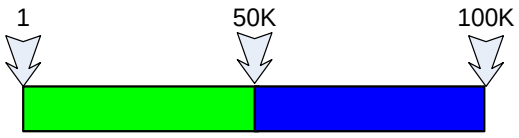

Рис. 6. Распределение нагрузки между потоками

Скомпилируйте и запустите приложение:

- В меню **Build** выберите команду **Build Solution**;
- В меню **Debug** выберите команду **Start Without Debugging**.

Убедитесь в правильности работы приложения по выводу на консоли.

3.6.2. Подготовка приложения для профилирования

Для того чтобы приложение можно было профилировать при помощи ITP, необходимо установить определенные настройки компиляции и компоновки проекта. Большинство из них установлены по умолчанию, но некоторые необходимо указывать самостоятельно.

1. В меню **Build** выберите пункт **Configuration Manager**, в открытом окне выберите режим **Release**.

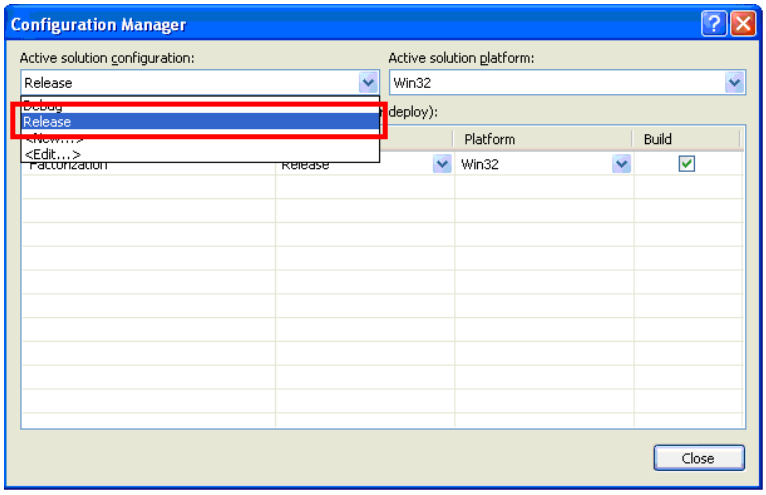

Рис. 7. Выбор режима Release

2. В меню **Project** выберите пункт **Properties**, в результате чего появится окно, представленное на рис. 8. В дереве слева выберите узел **Configuration Properties→C/C++→General**. В открывшейся таблице справа для элемента с именем **Debug Information Format** установите значение **Program Database (/Zi)**. Нажмите кнопку **Apply**.

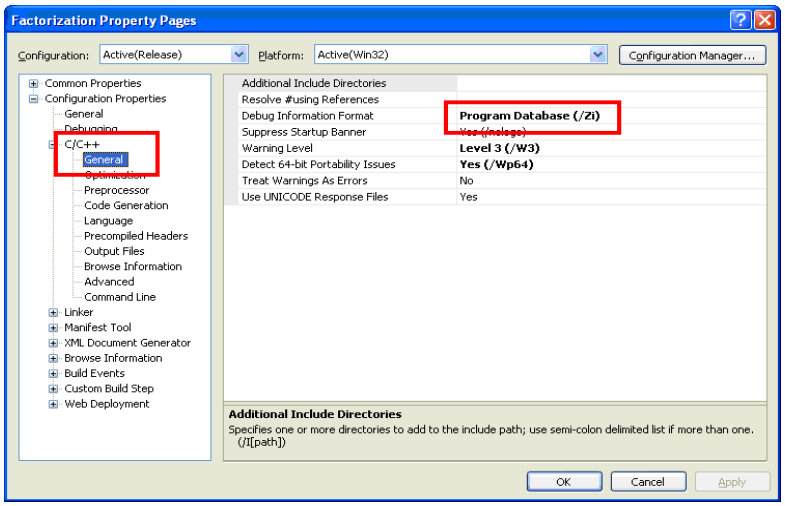

Рис. 8. Указание формата отладочной информации

3. В этом же окне настроек проекта необходимо убедиться в том, что для приложения генерируется отладочная информация. В дереве слева выберите узел **Configuration Properties→Linker→Debugging**. В открывшейся таблице справа для элемента с именем **Generate Debug Info** необходимо установить значение **Yes (/DEBUG)**. После этого нажмите кнопку **Apply**.

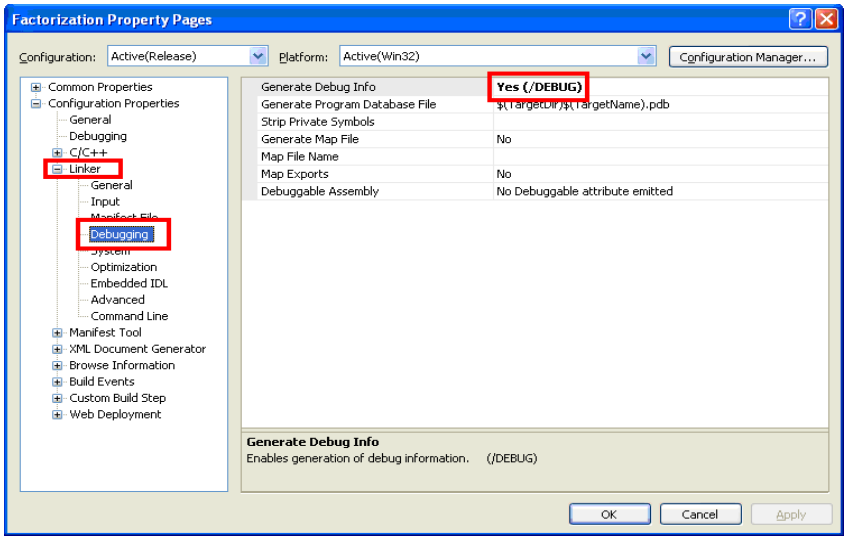

Рис. 9. Указание генерации отладочной информации

4. Убедитесь, что используются потокобезопасные библиотеки. Для этого выберите узел **Configuration Properties→C/C++→Code Generation**. В открывшейся таблице справа для элемента с именем **Runtime Library** установите значение **Multi-threaded DLL (/MD)**. Нажмите кнопку **Apply**.

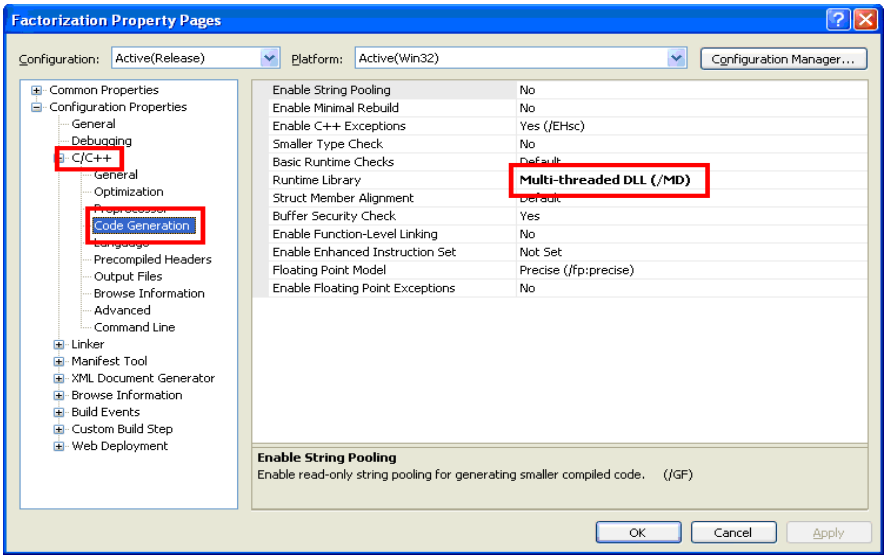

Рис. 10. Выбор потокобезопасных библиотек

5. Убедитесь, что приложение компонуется с использованием опции **/fixed:no**. В окне настроек проекта выберите узел **Configuration Properties→Linker→Advanced**. В открывшейся таблице справа для элемента с именем **Fixed Base Address** установите значение **Generate a relocation section (/FIXED:NO)**. Нажмите кнопку **Apply**.

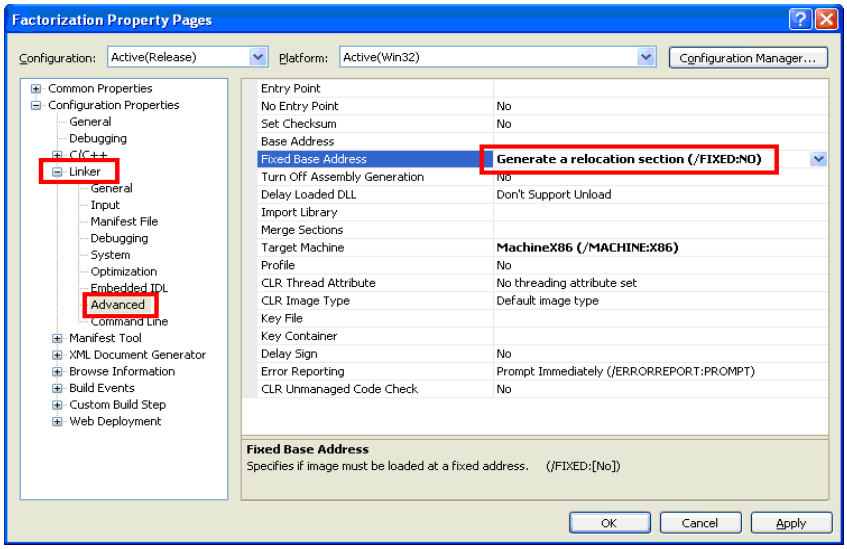

Рис. 11. Установка опций компоновщика

После выполнения данных процедур ваше приложение готово к профилированию.

3.6.3. Профилирование приложения

3.6.3.1. Создание проекта Intel Thread Profiler

1. Запустите Intel® Thread Profiler. Найти его можно, например, по следующему пути: **Start→All programs→Intel(R) Software Development Tools→Intel(R) Thread Profiler 3.0→Intel(R) Thread Profiler**.

2. В открывшемся окне нажмите на кнопку **New Project** .

3. В новом окне выберите **Intel(R) Thread Profiler Wizard** и нажмите кнопку **OK**.

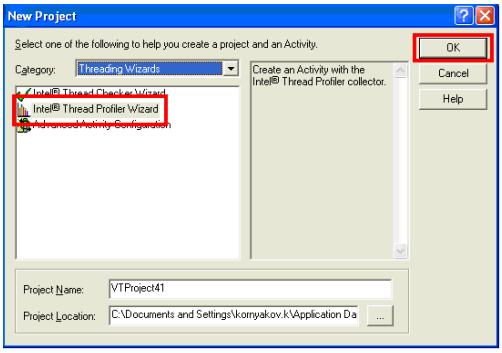

Рис. 12. Выбор типа проекта

4. Ниже надписи **Launch an application** имеется строка, в которой необходимо указать имя профилируемого приложения. Нажмите кнопку **[…]** и в открывшемся диалоговом окне укажите путь до приложения, подлежащего профилированию. В нашем случае это **C:\ITPLab\Factorization\release\Factorization.exe**.

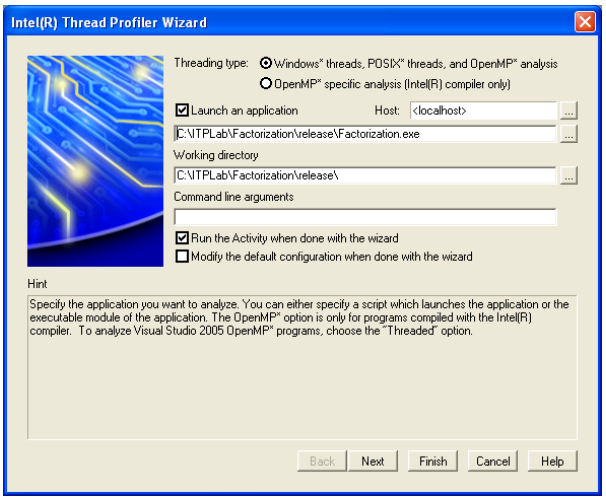

Рис. 13. Выбор профилируемого приложения

5. Нажмите кнопку **Finish.**

После этого ITP произведет инструментацию вашего приложения и немедленно начнет профилирование. Когда сбор трассы завершится, информация о ней появится на экране. Окно ITP содержит следующие элементы: панель инструментов, браузер результатов запусков (**Tuning Browser**), окно сообщений **Output** и окна **Profile** и **Timeline** (рис. 14). Последние два окна являются основными источниками информации о критическом пути приложения.

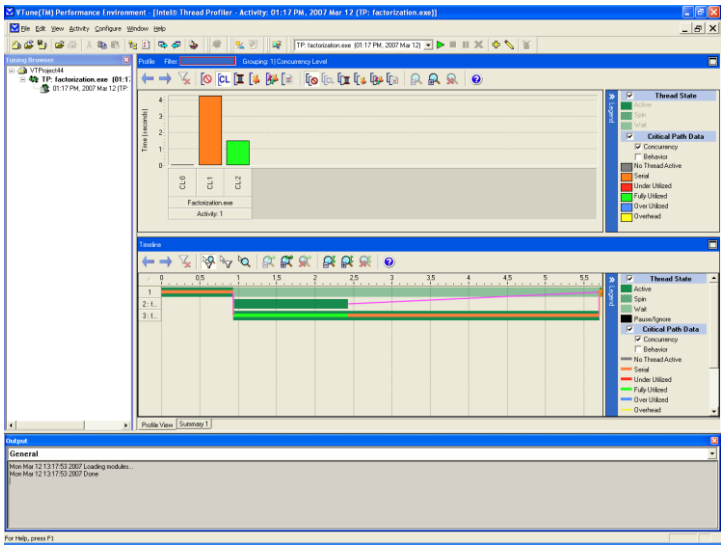

Рис. 14. Рабочая область Intel Thread Profiler

Профилирование

Профилирование приложения можно начать несколькими способами:

- Выбрать пункт меню **Activity→Run**;
- Нажать **F5**;
- Нажать на панели инструментов кнопку с зеленой стрелкой  .

Результаты каждого запуска сохраняются, и вы можете работать с несколькими профилями одновременно. Доступ к профилям осуществляется через пункт меню **View→Tuning Browser**.

3.6.4. Анализ производительности приложения

Далее мы по шагам изучим основные элементы графического интерфейса ITP, попутно рассматривая, как каждый из них используется при анализе производительности приложения.

Вкладка Summary

В нижней части окна ITP располагаются вкладки под названием **Profile View** и **Summary**. Перейдите на вторую из них, наведя на нее курсор и щелкнув левой кнопкой мыши. Эта вкладка содержит общую информацию о трассе приложения (рис. 15).

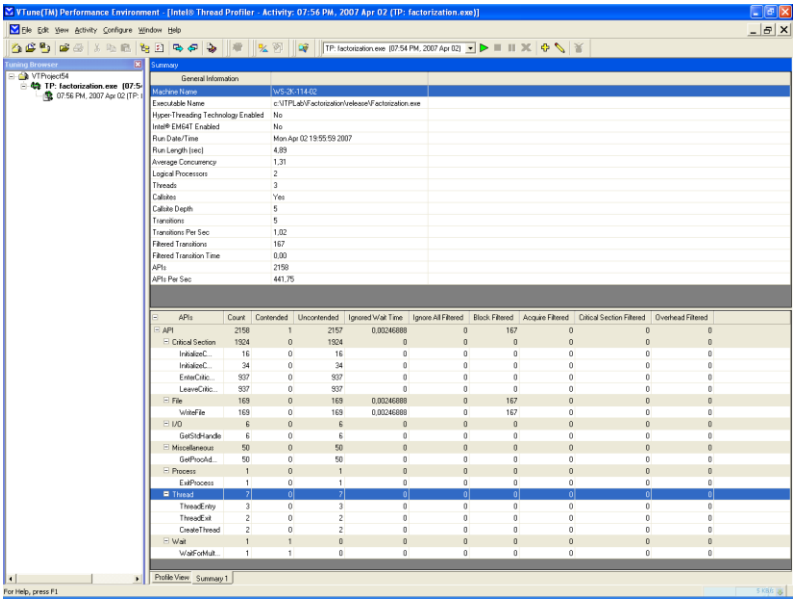

Рис. 15. Вкладка Summary

Вкладка состоит из двух таблиц: первая содержит общую информацию о состоявшемся запуске (характеристики узла, время работы приложения, число потоков и т.д.), а вторая – статистику вызовов API.

Щелкните на знак «+», располагающийся возле надписи «APIs» в нижней таблице, при этом она должна принять вид, показанный на рис. 15. Из таблицы

можно узнать, сколько времени приложение потратило на ввод/вывод, синхронизацию, непроизводительные издержки и т.д.

После того как вы ознакомитесь с приведенной информацией, вернитесь на вкладку **Profile View**.

Окно Profile

Основную часть окна **Profile** занимает область, на которой представлен критический путь приложения. Пользователь имеет возможность управлять представлением критического пути, для чего используется функция группировки и справка по использованию цветов (*легенда*), располагающаяся справа. Рассмотрим их подробнее.

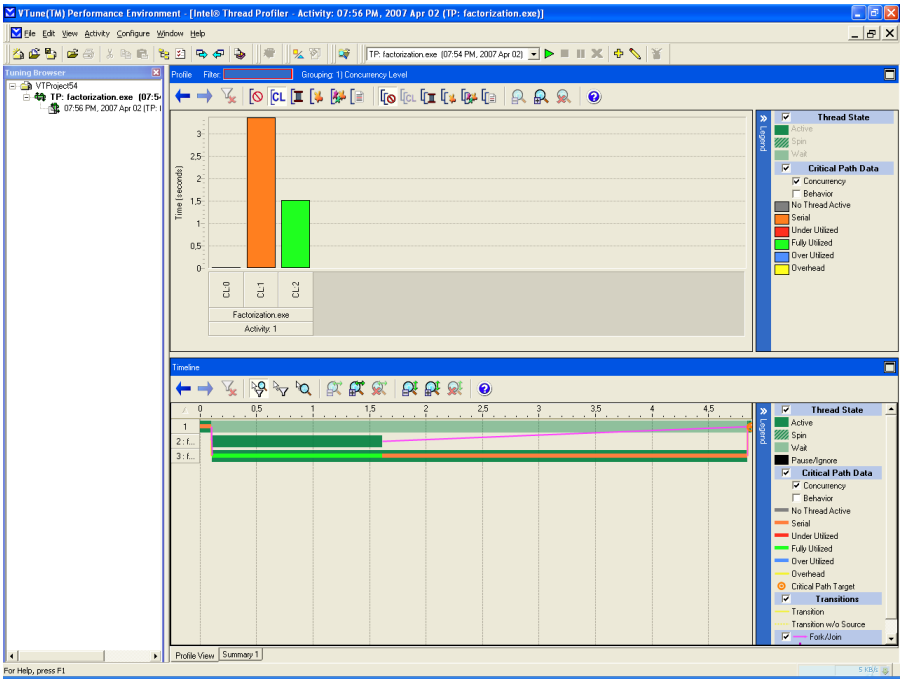

Рис. 16. Окно Profile

Выбор характеристик поведения потоков для визуализации

С помощью панели **Legend** пользователь имеет возможность указать интересующие его характеристики поведения потоков. Для этого используются находящиеся на легенде кнопки-флажки (check-box). Первый из них, называемый **Thread State**, позволяет указать, интересует ли нас информация о состоянии потока. Напомним, что существуют три типа состояния: *активное* (*active*), *активное ожидание* (*spin*), *ожидание* (*wait*).

Выберите тип группировки по потокам, нажав на панели кнопку с изображением катушки  . После этого в окне появится информация о распределении времени в критическом пути, как показано на рис. 17.

Информация о состоянии потоков представлена в виде столбиков зеленого цвета различной насыщенности. Бледно-зеленый цвет используется для обозначе-

ния того, что поток находился в состоянии ожидания, а темно-зеленый цвет соответствует активному состоянию потока. Из рисунка можно узнать, что два потока в нашем приложении (первый и третий столбики) большую часть времени были активны, а еще один поток в основном находился в состоянии ожидания. Нетрудно понять, что в ожидании находился основной поток нашего приложения, в то время как созданные им потоки производили разложение чисел на простые множители. Снимите флажок **Thread State** и убедитесь, что информация о состоянии потоков исчезнет. После этого верните флажок в исходное положение.

Следующая кнопка-флажок – **Critical Path Data**, при помощи которой пользователь может указать, интересует ли его разбиение критического пути по категориям времени. Снимите этот флажок и убедитесь, что на рисунке останется информация, касающаяся исключительно состояний потоков.

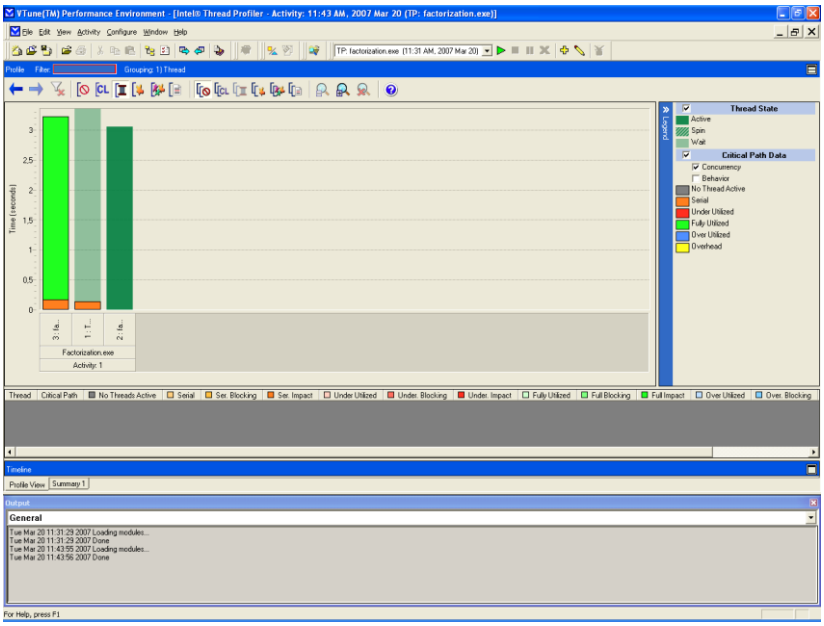

Рис. 17. Анализ состояний потоков

После этого верните все в исходное положение, в котором установлены флажки **Critical Path Data** и **Concurrency**, а флажок **Behavior** неактивен. В этой конфигурации легенды пользователь имеет возможность анализировать, насколько эффективно его приложение использует ядра процессора (*уровень параллелизма*). Здесь мы имеем дело с категориями времени, про которые было рассказано в разделе 3.3. В нашем случае можно увидеть, что основную часть критического пути занимает оранжевый цвет, что означает, что большую часть времени приложение выполнялось в последовательном режиме. Это тревожный симптом, который может означать недостаточно высокую степень распараллеливания или неравномерное распределение нагрузки между потоками.

Легенда также выполняет информационную функцию. Если вы забыли, что означает тот или иной цвет, наведите курсор мыши на прямоугольник соответствующего цвета в легенде. Появится всплывающая подсказка, содержащая информацию о том, что он означает. Кроме того, если цвет считается «проблемным»

(свидетельствует о неправильной организации приложения), то в подсказке будут содержаться советы для решения данной проблемы.

На легенде неизученной осталась последняя кнопка-флажок **Behavior**. Установите ее и снимите флажок **Concurrency**. В этой конфигурации пользователь имеет возможность определить поведение потока в его активном состоянии, о котором мы говорили ранее в разделе 3.3. В нашем случае критический путь окрашен в основном в красный цвет, что означает, что происходило ожидание завершения работы некоторых потоков. Это дает нам мало новой информации, поскольку мы и раньше знали, что основной поток приложения большую часть времени ожидает завершения дочерних потоков.

Установите флажки **Concurrency** и **Behavior**, чтобы получить комбинацию двух этих режимов. Тогда критический путь приобретет более разнообразную окраску. Предназначен этот режим для одновременного анализа всей совокупности характеристик поведения потоков.

Использование функции группировки

Данная функция очень удобна при анализе критического пути, она позволяет взглянуть на оптимизируемое приложение с различных точек зрения. Так, если осуществить группировку по потокам (кнопка ), то можно проанализировать эффективность работы каждого из потоков. Если же включить группировку по объектам (кнопка ), то становится доступной информация об эффективности работы с конкретным объектом (примитивом синхронизации, например).

Рассмотрим наиболее типичные варианты использования группировки. Полезно начать с варианта, когда группировки не используются. Нажмите кнопку с изображением перечеркнутого красного круга  – при этом критический путь примет вид, показанный на рис. 18.

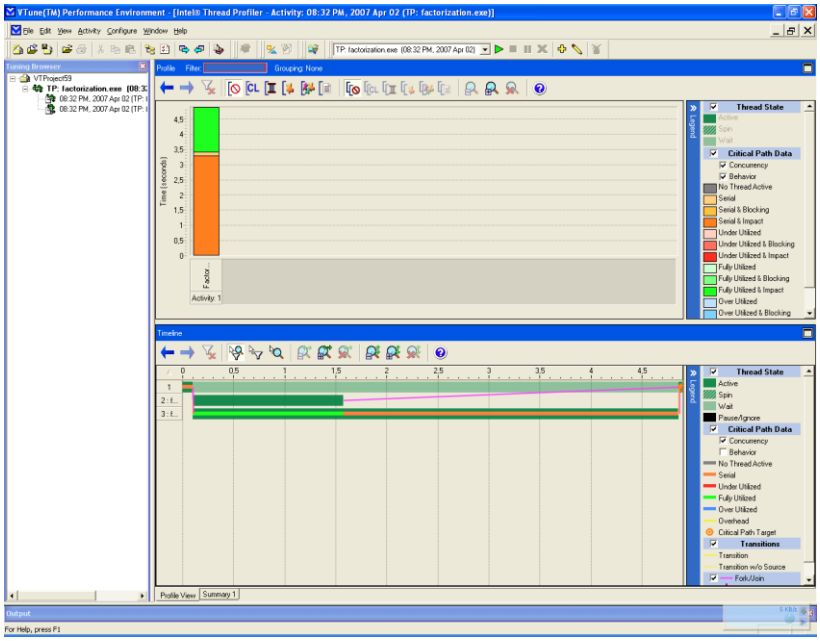

Рис. 18. Изображение критического пути при отключенной группировке

Этот режим наиболее полезен для получения информации о распределении времени в критическом пути. Используя его, можно получить информацию о том, сколько процентов занимает последовательное выполнение (доля оранжевого цвета), сколько параллельное (зеленый, синий и красный цвета), а также какую часть занимают непроизводительные расходы (доля желтого цвета). Попробуйте также другие режимы группировки, анализируя каждый раз изменения критического пути.

Кроме того, может оказаться полезной возможность вторичной группировки.

В первом наборе кнопок группировки нажмите кнопку с изображением катушки  , а во втором наборе – с буквами CL  . При этом критический путь примет вид, показанный на рис. 19.

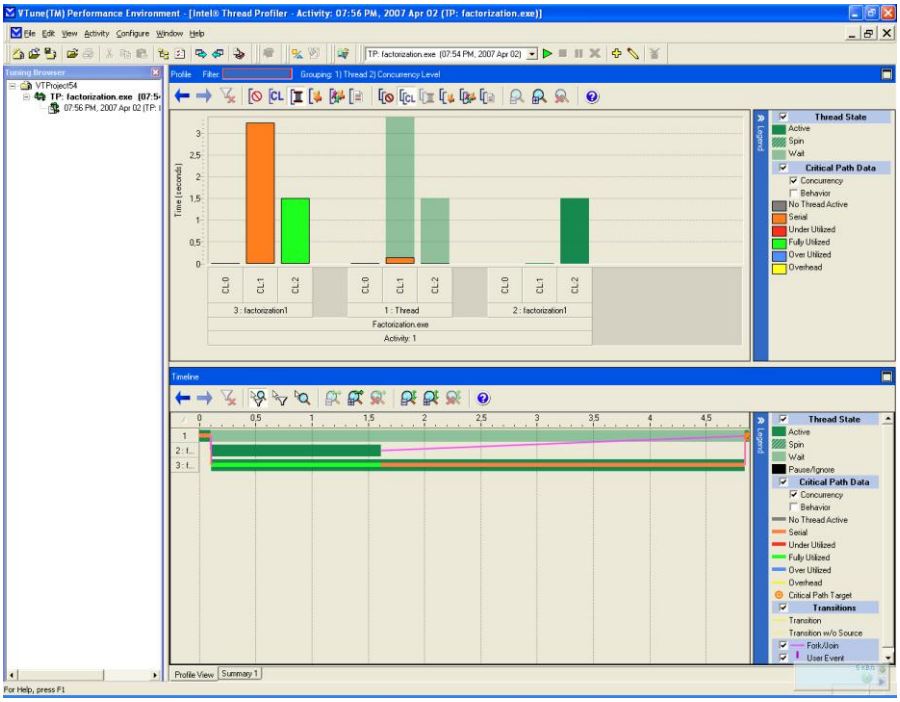

Рис. 19. Использование вторичной группировки

В данном случае мы имеем возможность исследовать, насколько эффективно функционировал каждый из потоков. Мы видим три группы столбиков, каждая из которых соответствует одному потоку. Столбики показывают, сколько времени каждый из потоков функционировал в последовательном режиме, в параллельном или в состоянии ожидания. Как можно увидеть из рисунка, второй поток (левая группа столбиков) почти все время работал параллельно. При этом третий поток (левая группа столбиков) не менее половины времени работал в последовательном режиме. Основной поток приложения (центральная группа столбиков) выполнялся лишь в последовательном режиме и большую часть жизни находился в режиме ожидания дочерних потоков.

Поэкспериментируйте, определяя различные порядки группировки, попытайтесь представить ситуации, в которых они могут быть полезны.

Рабочая область

Мы изучили различные способы представления критического пути в рабочей области окна **Profile,** и пришло время изучить способы его анализа.

Выключите группировку, чтобы распределение критического пути по категориям времени было представлено в виде одного столбца, как на рис. 20. Кроме того, установите все флажки на легенде. После этого наведите курсор на каждый из участков критического пути с различными цветами. В появляющихся подсказках содержится информация об участке критического пути, на который указывает курсор мыши.

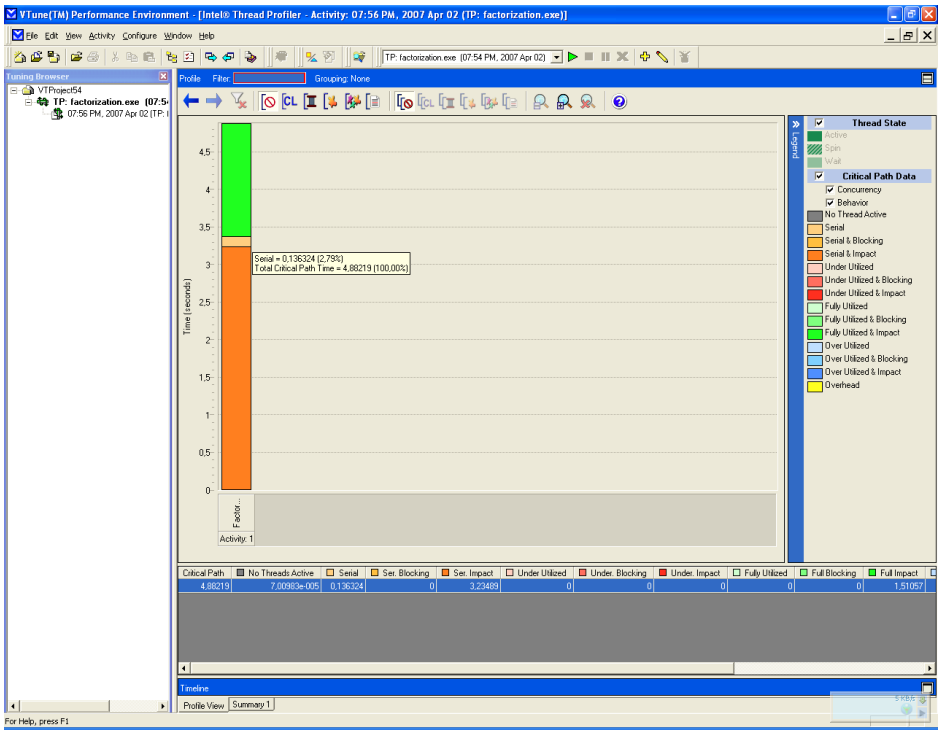

Рис. 20. Анализ критического пути

Наведите курсор мыши на столбец и щелкните левой кнопкой один раз. В нижней части окна **Profile** появится информация о том, сколько времени приложение функционировало в различных режимах – распределение времени по категориям.

Двойной щелчок на столбце позволяет получить более подробную информацию. Попробуем, например, получить полную информацию об участке критического пути оранжевого цвета (здесь и далее имеется в виду ярко-оранжевый цвет). Для этого наведите на него курсор и произведите двойной щелчок левой кнопкой мыши. Критический путь распадется на уровни параллелизма (*concurrency*), в чем можно убедиться по нажатой кнопке CL на панели группировки и надписям под различными столбцами. Щелкните по оранжевому участку еще раз – и единый столбец снова распадется на несколько, уже по числу потоков. Еще один двойной щелчок приведет нас к уровню функций. Теперь надпись под столбцом однознач-

но указывает на имя функции, время работы которой вошло в критический путь с оранжевым цветом. Если произвести еще один двойной щелчок, то откроется вид исходного кода (рис. 21).

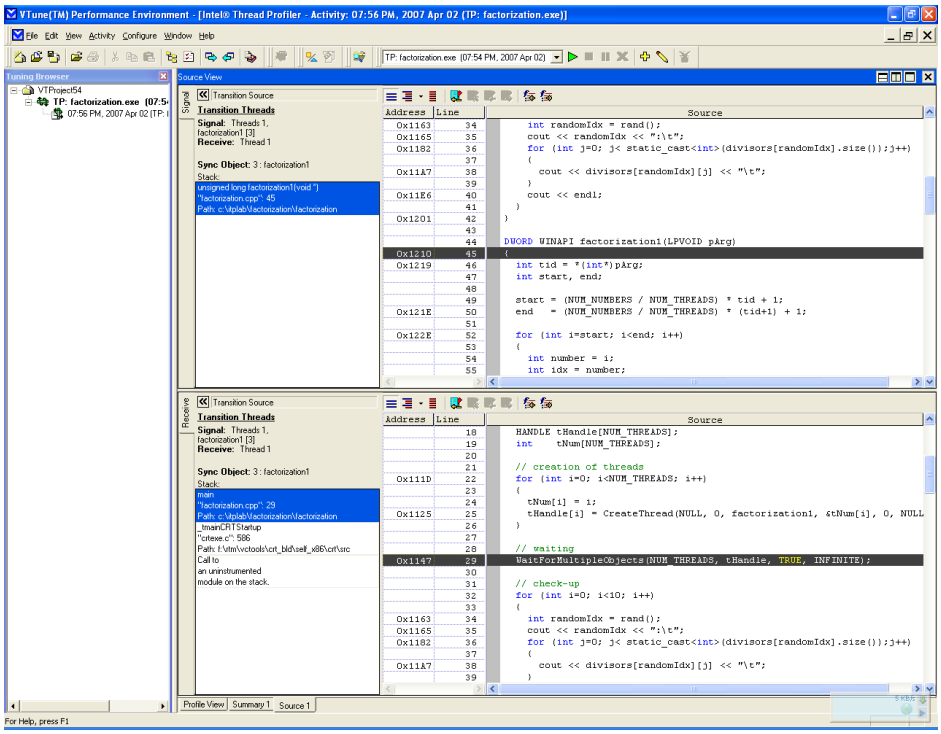

Рис. 21. Обращение к исходному коду приложения

Здесь мы можем убедиться, что за участок критического пути оранжевого цвета отвечает рабочая функция потоков, производящих факторизацию чисел.

Чтобы вернуться к исходному представлению критического пути, выключите все фильтры, нажав кнопку с изображением воронки  , а затем отмените всякую группировку, нажав кнопку  .

Если имеется необходимость понять, какой участок кода отвечает за некоторый участок критического пути, можно поступить более простым способом, чем мы делали это раньше. А именно, выберите способ группировки по исходному коду. Для этого нажмите на кнопку с изображением файла  . После этого наведите курсор мыши на интересующий вас участок критического пути и нажмите правую кнопку мыши. Если ITP имеет доступ к коду, который отвечает за данный участок критического пути, то в контекстном меню будет доступен пункт **Transition Source View**. Выберите его – откроется вид исходного кода.

Однако не всегда можно спуститься до уровня исходного кода – так случается, если в приложении не содержится отладочная информация. Типичная ситуация – приложение использует вызовы библиотеки, которая была получена в скомпилированном виде.

Итак, окно **Profile** предоставляет возможности анализа критического пути. С его помощью можно узнать распределение времени работы приложения по кате-

гориям. Но при этом мы получаем очень мало информации о динамике работы приложения. Для понимания того, что происходило с потоками во время исполнения, как они взаимодействовали между собой, обратимся к следующему окну ITP – окну **Timeline**.

Окно Timeline

Окно **Timeline** содержит три основных элемента, так же как и окно **Profile**: рабочая область, панель инструментов и легенда. Панель инструментов в данном случае предназначена лишь для изменения масштаба временной оси.

Рабочая область

В рабочей области окна **Timeline** содержится информация о поведении потоков. В верхней части рабочей области имеется временная шкала. Ниже располагаются полосы, каждая из которых соответствует одному потоку приложения.

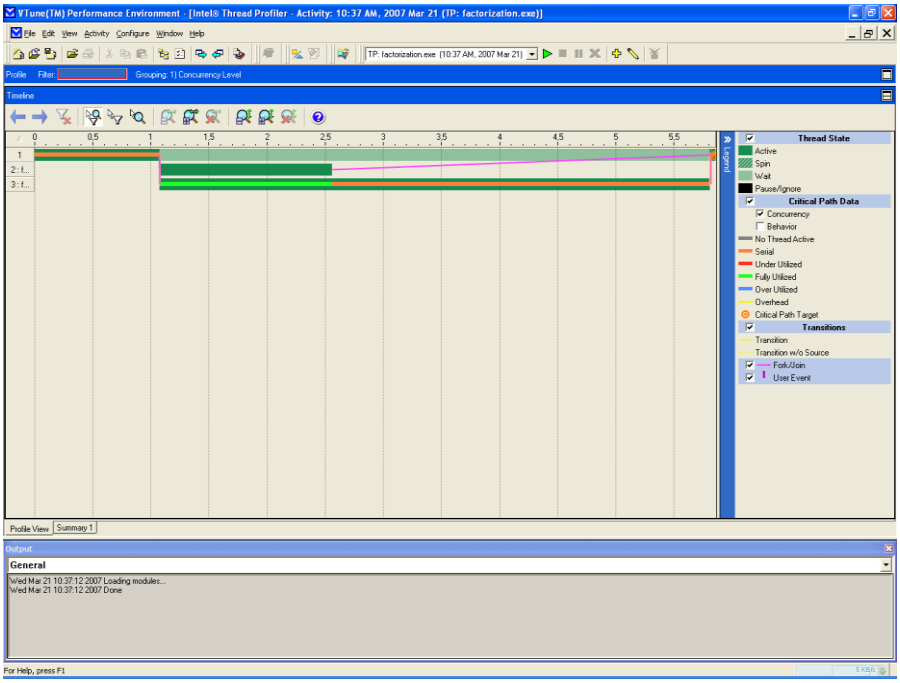

Рис. 22. Окно Timeline

На рис. 22 показано, что в нашем приложении было запущено три потока. Также можно увидеть, что один из них существовал с начала времени выполнения приложения, а два других были созданы позже. Время жизни потока равно длине полосы. Раскраска полосы не всегда одинакова – она указывает на состояния потока в различные моменты времени (см. легенду). Наведите курсор мыши на бледно-зеленую часть самой верхней полосы, соответствующей основному потоку. В появившейся подсказке будет сказано, что на этом промежутке времени поток находился в режиме ожидания.

Рассмотрим подробнее момент возникновения потоков. Для этого в рабочей области необходимо выделить область, охватывающую розовую стрелку. Наведите курсор мыши немного левее стрелки, зажмите левую клавишу и передвиньте курсор правее стрелки, после чего отпустите кнопку мыши. Повторите увеличение еще несколько раз, пока вид в рабочей области не станет таким, как показано на рис. 23.

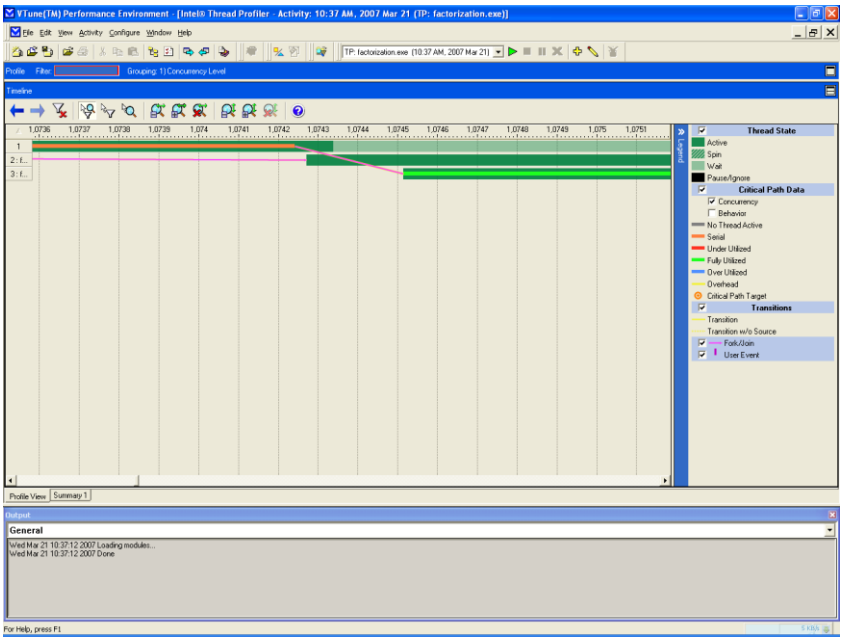

Рис. 23. Момент создания потоков

Здесь уже можно видеть, что второй поток был создан несколько позже первого. Наведите курсор мыши на одну из розовых стрелок и убедитесь, что она соответствует вызову функции создания потока. Длина этой стрелки, спроектированная на ось времени, показывает время задержки между вызовом функции создания потока и фактическим его запуском.

Кроме того, поверх зеленых полос, указывающих состояние потоков, яркими цветами накладывается информация о критическом пути. На изучаемом нами промежутке времени имеются участки последовательного и параллельного исполнения потоков. Нетрудно понять, что оранжевый участок соответствует интервалу времени, когда в приложении существовал только один поток, а участок зеленого цвета соответствует одновременной работе двух потоков.

Вернемся к изучению трассы приложения в целом. Нажмите на панели инструментов окна **Timeline** кнопку с изображением воронки . Рабочая область снова должна принять вид как на рис. 22.

Глядя на этот рисунок, можно сразу понять, в чем причины того, что основную часть времени наше приложение работает в последовательном режиме. Мы неравномерно распределили вычислительную нагрузку между потоками. Второй из дочерних потоков работает гораздо дольше первого, в результате чего увеличивается суммарное время работы приложения. Таким образом, если мы хотим повысить производительность нашего приложения, то первое, что мы должны сде-

лать, – распределить нагрузку между потоками равномерно. Тогда время работы приложения сократится за счет того, что часть нагрузки второго дочернего потока будет отдана первому.

Последнее, что осталось изучить в рабочей области окна **Timeline** – это как из него получить доступ к исходному коду. Наведите курсор мыши на любую из полос, соответствующих дочерним потокам, и нажмите правую кнопку мыши. В появившемся контекстном меню выберите пункт **Thread Creation/Entry Source View**. Откроется окно с исходным кодом, отвечающим за создание потока.

Вернитесь к окну **Timeline** и наведите курсор на бледно-зеленую часть полосы, соответствующей основному потоку приложения, нажмите правую кнопку мыши и выберите пункт **Transition Source View**. Откроется окно с исходным кодом, в котором указано место ожидания основного потока (рис. 24).

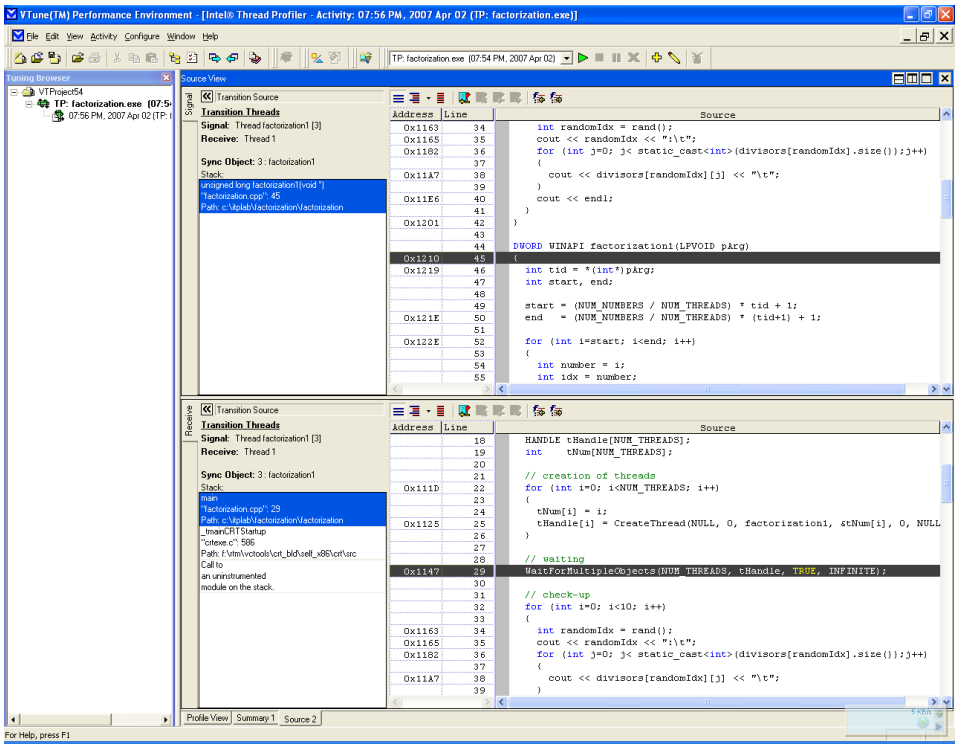

Рис. 24. Место ожидания основного потока

В нашем случае это вызов функции **`WaitForMultipleObjects`**, которая необходима здесь для ожидания завершения дочерних потоков.

Вернемся к окну **Timeline** и рассмотрим легенду.

Выбор показателей поведения потоков для визуализации

Общая структура и функции легенды такие же, как и в окне **Profile**, поэтому мы не будем останавливаться на кнопках-флажках **Thread State** и **Critical Path Data**. Вы можете поэкспериментировать с ними и проследить, как меняется вид в рабочей области окна **Timeline**.

Рассмотрим назначение новых кнопок-флажков. Первый из них имеет название **Transitions** и отвечает за отображение в рабочей области стрелок, соответствующих посылке сигналов между потоками. Снимите флажок **Fork/Join** – останутся три стрелки желтого цвета, показывающие посылаемые в приложении сигналы. В нашем случае это сигналы, связанные с созданием и завершением потоков.

Теперь снимите флажок **Transitions** и установите флажок **Fork/Join**. Должны появиться стрелки, указывающие на вызовы функций класса **`Fork/Join`** (создание и ожидание завершения потоков). Можно заметить, что они совпадают с желтыми стрелками, которые мы видели до этого. Единственное отличие – появление розовой стрелки, соединяющей конец полосы первого дочернего потока с полосой основного потока.

Последний флажок называется **User Event**. Мы не станем рассматривать его, подробная информация о нем может быть найдена в [7].

Таким образом, мы выяснили, что наше приложение содержит поток, большую часть времени работающий в последовательном режиме. Причины такой ситуации и способы увеличения производительности нашего примера мы рассмотрим в лабораторной работе 1, раздел 4.

3.7. Контрольные вопросы

1. Наличие каких цветов в критическом пути свидетельствует о проблемах с производительностью приложения?
2. На какие цвета, по вашему мнению, нужно обратить внимание прежде всего?
3. Если вам известно несколько причин низкой производительности вашего приложения, в какой последовательности лучше их устранять?
4. Зачем используется группировка?
5. В чем разница между группировкой по объектам и по типам объектов?
6. В окне **Profile** установите первичную группировку по потокам, а вторичную – по уровню параллелизма. Установите соответствие между столбцами в окне **Profile** и полосами в окне **Timeline**.
7. Как было установлено, основная причина медленной работы учебного приложения – неравномерное распределение нагрузки между потоками. Предложите ваш способ решения данной проблемы.

3.8. Литература

1. Developing Multithreaded Applications: A Platform Consistent Approach. Intel Corporation, March 2003.
2. Threading Methodology: Principles and Practices. Intel Corporation, March 2003.
3. Multi-Core Programming for Academia. Student Workbook, by Intel.
4. Akhter Sh., Roberts J. Multi-Core Programming. Intel Press, 2006.
5. Intel® Thread Profiler. Getting Started Guide.
6. Intel® Thread Profiler. Guide to Sample Code.
7. Intel® Thread Profiler Help.

4. Лабораторная работа. Профилирование параллельной программы с использованием Intel Thread Profiler. Балансировка нагрузки

4.1. Цели лабораторной работы

Изучить основные вопросы, возникающие при распределении нагрузки между потоками. Рассмотреть различные подходы к балансировке, приобрести навыки их анализа и сравнения между собой при помощи Intel® Thread Profiler (ITP).

4.2. Инструкция для выполнения лабораторной работы

Предполагается, что в процессе начального ознакомления с ITP вы уже изучили приложение **Factorization**, производящее разложение множества чисел на простые множители (факторизация). Подробное описание приложения может быть найдено в [8].

Как мы установили ранее, основная причина низкой производительности приложения **Factorization** заключается в неравномерном распределении нагрузки между потоками, производящими факторизацию. Далее мы последовательно рассмотрим несколько различных подходов к балансировке и попытаемся определить, какой из них является наиболее эффективным для нашего приложения.

Замечание: результаты, которые будут получены вами, могут несколько отличаться от приведенных далее. Повлиять на это могут характеристики вычислительного узла, на котором выполняется лабораторная работа, прежде всего количество ядер. Мы производили профилирование на двухъядерной машине.

4.2.1. Подход 1: разделение множества чисел на одинаковые части по числу потоков

Этот подход уже был рассмотрен нами ранее. Мы, однако, обратимся к нему еще раз, чтобы понять, в чем причины дисбаланса нагрузки между потоками. Одновременно с этим попытаемся найти более эффективный способ распределения множества чисел между потоками.

Откройте файл **Factorization.cpp** в проекте **Factorization**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**…;
- в диалоговом окне **Open Project** выберите папку **C:\ITPLab\Factorization**;
- дважды щелкните на файле **Factorization.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **Factorization.cpp**. Сейчас нас интересует рабочая функция потока под названием `factorization1`. Она реализует следующую стратегию распределения нагрузки: первый поток получает множество чисел от 1 до 50000, а второй – от 50001 до 100000.

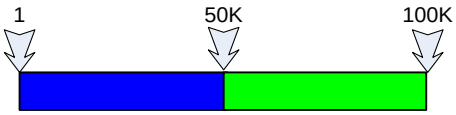

Рис. 1. Разделение множества чисел на одинаковые части по числу потоков

Убедитесь, что в программе в качестве аргументов функции **`CreateThread`** передается именно функция **`factorization1`**. В дальнейшем, чтобы использовать другой подход к распределению нагрузки, необходимо будет лишь заменить имя рабочей функции потока на **`factorization2`** или **`factorization3`**.

Далее необходимо подготовить приложение к профилированию. Сделайте это, как указано в описании [8], после чего создайте проект в ITP и запустите процесс профилирования. Рабочая область ITP должна принять вид, как показано на рис. 2.

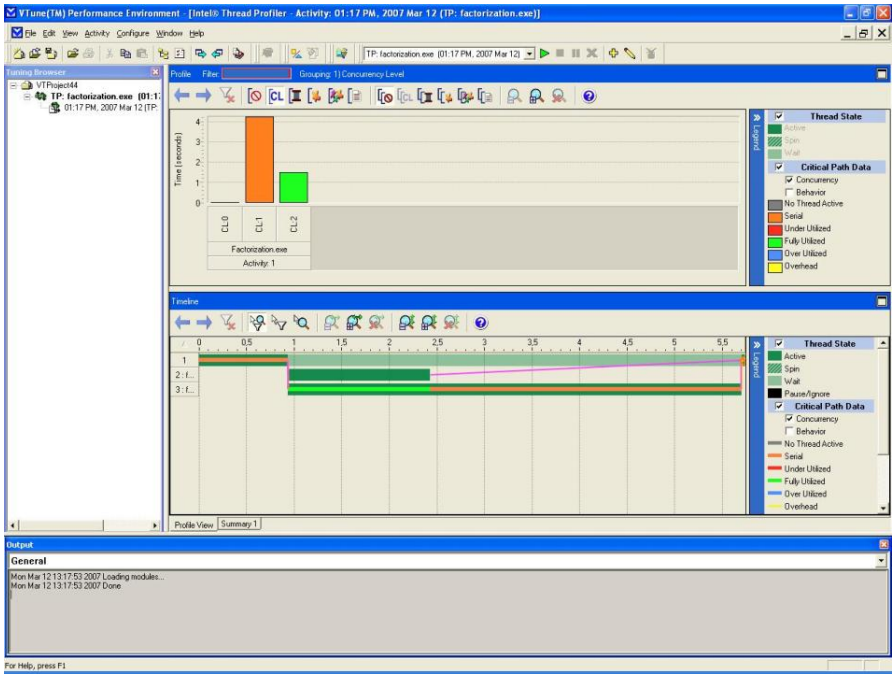

Рис. 2. Результат профилирования приложения, использующего первый подход к распределению нагрузки

В окне **Profile** можно заметить, что столбик оранжевого цвета оказался выше других. Это означает, что большую часть времени приложение выполнялось в последовательном режиме, то есть работал лишь один поток из трех. Далее обратите внимание на окно **Timeline**. После взгляда на него становится ясно, в чем причина преобладания последовательного исполнения. Неравномерное распределение нагрузки привело к тому, что второй дочерний поток работает гораздо дольше первого и тормозит работу приложения в целом.

Недостаток выбранного подхода распределения нагрузки очевиден: первый поток факторизует числа от 1 до 50000, что существенно проще, чем разложить числа от 50001 до 100000. Итак, наша задача состоит в том, чтобы придумать более эффективный способ разделения чисел между потоками.

4.2.2. Подход 2: разделение множества чисел на четные и нечетные

Рассмотрим следующий способ распределения нагрузки. Проще всего сделать так, чтобы потоки брали числа по очереди. То есть в нашем случае первому потоку достанутся все нечетные числа, а второму все четные. На рис. 3 синим цветом помечены числа, которые будет обрабатывать первый поток, а зеленым – второй.

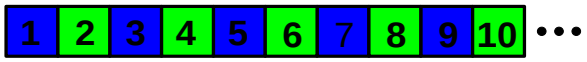

Рис. 3. Разделение множества чисел на четные и нечетные

Описанный подход реализован в функции **`factorization2`**. Ознакомьтесь с ней и в месте вызова функции **`CreateThread`** замените ее аргумент **`factorization1`** на **`factorization2`**, чтобы приложение стало использовать разбиение множества чисел на четные и нечетные.

Перекомпилируйте приложение и запустите процесс профилирования. Сделайте это в рамках того же проекта в ITP, чтобы в окне **Tuning Browser** сохранились результаты предыдущего запуска и мы могли впоследствии сравнить их с новыми результатами. Рабочая область ITP должна принять вид, как показано на рис. 4.

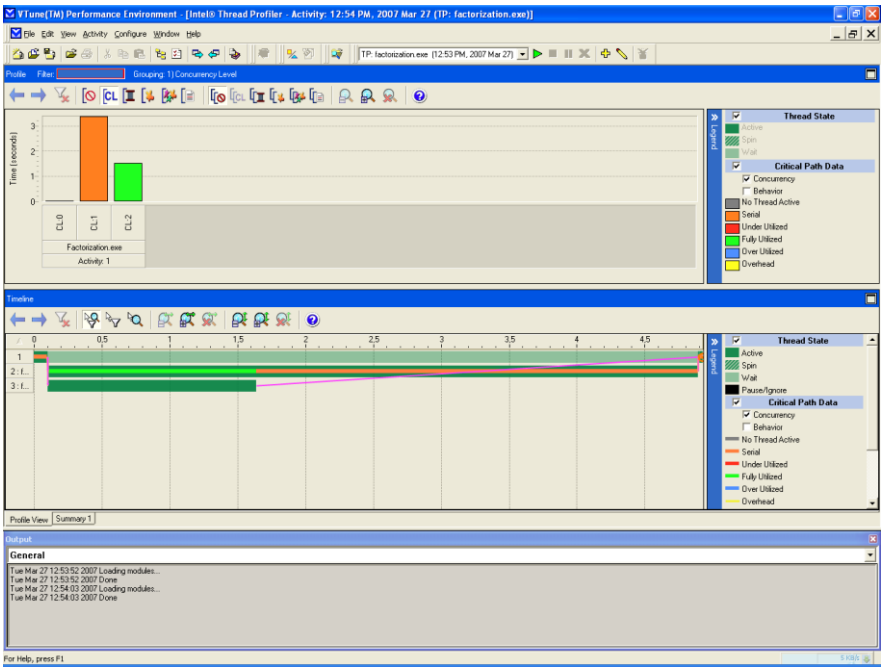

Рис. 4. Результат профилирования приложения, использующего второй подход к распределению нагрузки

Результаты окажутся неожиданными на первый взгляд. Высота оранжевого столбика практически не изменилась, но потоки как будто поменялись местами. Теперь первый из дочерних потоков выполняется гораздо медленнее второго.

Причина и в этот раз находится на поверхности. Четные числа факторизовать существенно проще, чем нечетные, поскольку после первого же деления (на число

2) их величина сокращается в два раза. Мы снова сталкиваемся с ситуацией, когда один поток вынужден совершать работу гораздо более трудоемкую, чем второй.

Очевидно, и этот подход дает неудовлетворительные результаты.

4.2.3. Подход 3: разделение множества чисел на небольшие пачки

Итак, мы уже рассмотрели два подхода к балансировке нагрузки, но они оказались неэффективными из-за специфики нашей задачи. Нам нужен способ, позволяющий более равномерно нагрузить потоки.

Существует довольно популярный метод в задачах обработки множества однотипных заявок – обработка их целыми пачками. Все множество заявок разделяется на множество пачек небольшой длины, после чего эти пачки распределяются поровну между потоками. Это изображено схематично на рис. 5 (размер пачки равен 1000).

Рис. 5. Разделение множества чисел на небольшие пачки

Реализован этот подход в функции **`factorization3`**. Ознакомьтесь с ее кодом и перекомпилируйте приложение, подставив **`factorization3`** в качестве аргумента функции **`CreateThread`**. Перейдите обратно в ITP и запустите процесс профилирования. При этом должны появиться графики, представленные на рис. 6.

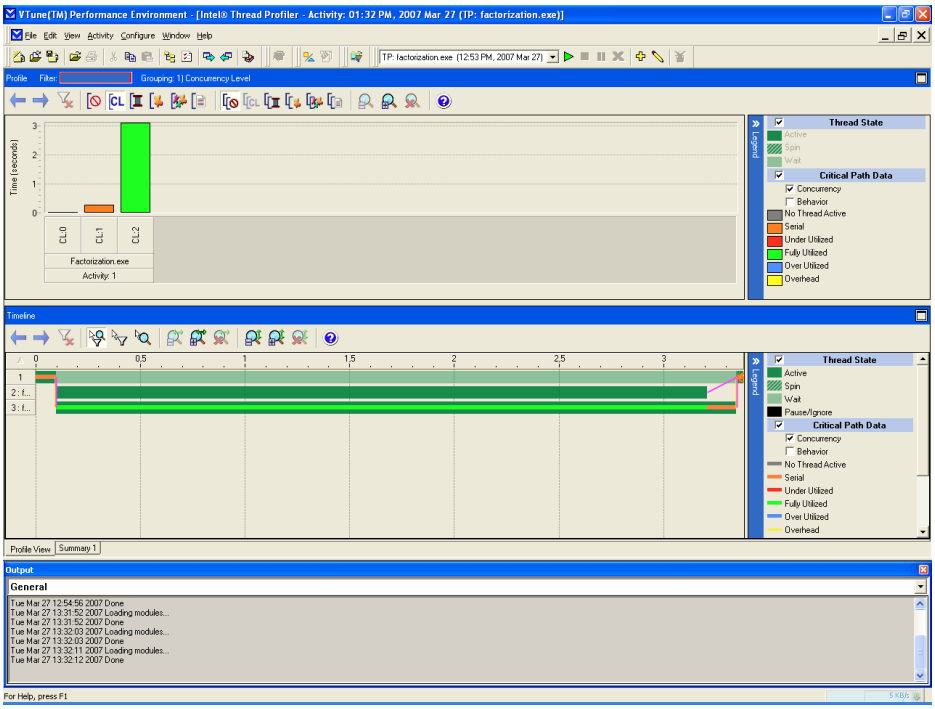

Рис. 6. Результат профилирования приложения, использующего третий подход к распределению нагрузки

В этот раз мы достигли успеха. Основную часть времени приложение выполняется в параллельном режиме, а время работы сократилось с 5 до 3,5 секунд, что означает увеличение производительности на 30%!

Итак, нам все-таки удалось подобрать верную стратегию распределения нагрузки между потоками.

4.3. Самостоятельная работа. Выбор оптимальной степени гранулярности

Под *гранулярностью* понимают детальность разбиения исходной задачи на меньшие подзадачи. В нашем случае степень гранулярности означает размер пачки чисел, которые мы используем в третьем подходе распределения нагрузки. За этот параметр отвечает константа **`GRAIN_SIZE`**, которая объявлена непосредственно перед кодом реализации функции **`factorization3`**.

Определение оптимальной степени гранулярности – самостоятельная задача в некоторых приложениях. В нашем случае если размер пачки очень велик (например, 50000), то подход выродится в первый из рассмотренных нами. Слишком мелкое дробление исходной задачи тоже неэффективно, особенно если требуется синхронизация между потоками после обработки очередной пачки (сохранение промежуточных результатов в глобальные переменные). В некоторых случаях время на синхронизацию может даже превышать выигрыш от параллельной обработки, поэтому степень гранулярности нужно выбирать очень осторожно. В нашем приложении синхронизации не производится, но явление снижения производительности при малом размере пачки все равно имеет место.

Путем экспериментов определите оптимальную степень гранулярности. Попытайтесь найти объяснение, почему именно найденная вами величина дает максимальную производительность.

4.4. Контрольные вопросы

1. В задачах какого класса должен хорошо показать себя первый из описанных подходов к распределению нагрузки?
2. Как вы думаете, если использовать второй подход с четырьмя потоками, равномерно ли распределится нагрузка между ними? Почему? Проведите эксперимент и проверьте свои предположения.
3. Предложите свой метод распределения нагрузки между потоками. Если имеется время, запрограммируйте его и сравните с подходами, приведенными в настоящем документе.
4. Как вы думаете, какой должен быть следующий шаг, если мы хотим продолжить повышение производительности приложения **Factorization**?

4.5. Литература

1. Developing Multithreaded Applications: A Platform Consistent Approach. Intel Corporation, March 2003.
2. Threading Methodology: Principles and Practices. Intel Corporation, March 2003.
3. Multi-Core Programming for Academia. Student Workbook, by Intel.
4. Akhter Sh., Roberts J. Multi-Core Programming. Intel Press, 2006.
5. Intel® Thread Profiler. Getting Started Guide.
6. Intel® Thread Profiler. Guide to Sample Code.
7. Intel® Thread Profiler Help.
8. Intel® Thread Profiler. Краткое описание. Материалы по образовательному комплексу «Технологии разработки параллельных программ».

5. Лабораторная работа. Профилирование параллельной программы с использованием Intel Thread Profiler. Синхронизация и накладные расходы на поддержку многопоточности

5.1. Цель лабораторной работы

Целью настоящей лабораторной работы является изучение способов снижения накладных расходов на поддержку многопоточности. Под накладными расходами понимаются непроизводительные издержки на работу с потоками (время их создания, управления и уничтожения), а также время, затрачиваемое на работу с примитивами синхронизации (например, время с момента посылки сигнала до его получения). Кроме того, рассматриваются вопросы, связанные с выбором и использованием примитивов синхронизации.

5.2. Инструкция для выполнения лабораторной работы

5.2.1. Изучение профилируемого приложения

В настоящей лабораторной работе используется учебное приложение **ClientServer**, которое имитирует работу системы, имеющей клиент-серверную архитектуру. Особенность состоит в том, что для простоты клиент и сервер не представляют собой различные процессы, а реализованы в виде потоков одного процесса. Сценарий приложения следующий: клиент посылает запросы нескольких типов, а сервер принимает и обрабатывает их. Обработка производится в специально создаваемых для этого потоках.

Откройте проект **ClientServer**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**…;
- в диалоговом окне **Open Project** выберите папку **C:\ITPLab\ClientServer;**
- дважды щелкните на файле **ClientServer.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** выберите проект **ClientServer** и дважды щелкните на файле исходного кода **ClientServer.cpp**, как это показано на рис. 1. После этих действий программный код, с которым предстоит работать, будет открыт в рабочей области **Microsoft Visual Studio**.

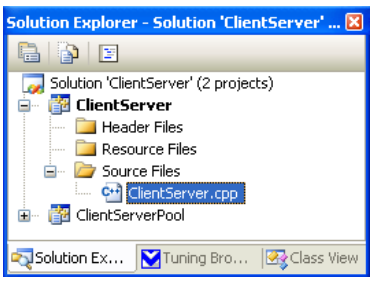

Рис. 1. Открытие файла ClientServer.cpp

В начале файла **ClientServer.cpp** объявлены две константы:

```
const int numRequests = 100;
const int numTypes = 4;
```

Первая из них указывает количество запросов от клиента к серверу, вторая показывает, сколько существует различных типов запросов. В данном случае клиент передаст серверу 100 запросов четырех типов. Структура запроса в нашем приложении предельно проста – это целое число от 0 до **`numTypes`**. То есть клиент каждый раз посылает серверу число, которое и означает тип пришедшего запроса.

Кроме того, в приложении как глобальные переменные объявлены очередь и вектор:

```
queue<int>  requests;           //queue for requests
vector<int> requestsStatistics; //requests statistics
```

Очередь используется для передачи запросов от клиента к серверу, а вектор – для сохранения статистики о пришедших запросах. Так, например, значение **`requestsStatistics[0]`** показывает количество пришедших запросов типа 0.

Ознакомьтесь с кодом приложения. Как можно видеть, для обработки каждого запроса сервер создает новый поток. Таким образом, приложение состоит из трех компонентов:

- функция **`main`** – рабочая функция потока-клиента;
- функция **`ServerThreadFunc`** – рабочая функция потока-сервера;
- функция **`HandlerPoolThreadFunc`** – рабочая функция потока-обработчика.

Скомпилируйте и запустите приложение стандартными средствами **Microsoft Visual Studio**:

- щелкните правой кнопкой мыши на проекте **ClientServer** и выберите в контекстном меню пункт **Build Only ClientServer** (рис. 2);
- в этом же меню выполните команду **Debug→Start new instance**.

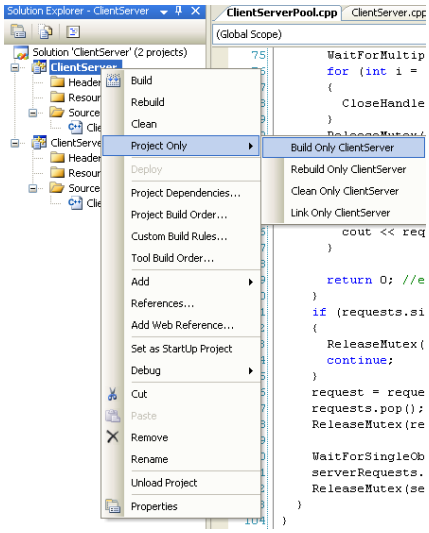

Рис. 2. Компиляция приложения

Убедитесь в правильности работы приложения по выводу на консоль.

5.2.2. Профилирование приложения

Запустите процесс профилирования в ITP, в его окне должны появиться результаты, как показано на рис. 3.

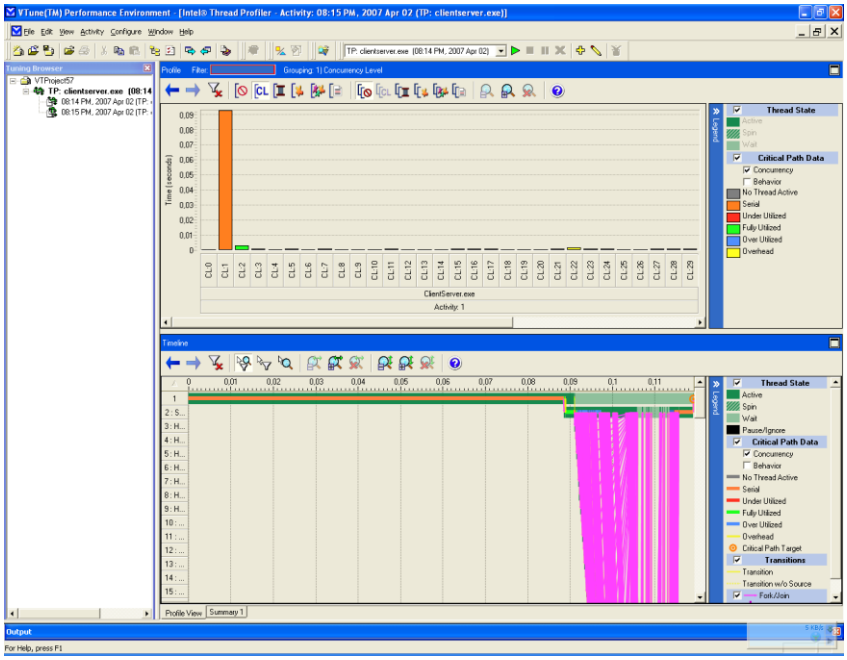

Рис. 3. Результаты профилирования приложения

Обратите внимание на область розового цвета в окне **Timeline**. При увеличении видим, что она образована стрелками, соответствующими вызовам **`fork/join`**. Это тревожный симптом, свидетельствующий о том, что в нашем приложении создается слишком много потоков. Ситуация эта плоха тем, что полезная работа, которую совершают потоки, не компенсирует затрат на их создание/уничтожение.

Действительно, для каждого запроса сервер создает новый поток, поэтому в нашем приложении существует **`numRequests`**+2 потока (сервер и клиент). Наш сервер, создавая и уничтожая потоки, тратит больше времени и потребляет больше системных ресурсов, чем если бы обрабатывал запросы самостоятельно. Кроме того, активные потоки также потребляют системные ресурсы, что может привести к нехватке оперативной памяти и значительному падению производительности.

Вообще говоря, работа многих серверов (web-серверы, серверы базы данных) связана с обработкой большого количества коротких запросов от какого-либо удаленного источника (клиента). При этом существуют несколько распространенных вариантов архитектуры многопоточного сервера:

- обработка всех запросов в одном потоке;
- обработка каждого нового запроса в отдельном потоке;
- организация пула потоков.

Рассмотрим более подробно эти варианты.

5.2.2.1. Обработка всех запросов в одном потоке

Это решение подходит для тех случаев, когда количество запросов к серверу достаточно мало и обращения к серверу происходят редко. При этом архитектура приложения очень проста, единственной трудностью является построение очереди входящих запросов, необходимой для предотвращения потери запросов при последовательной обработке.

Однако этот подход крайне неэффективен при высокой частоте обращений к серверу. Если сервер производит обработку самостоятельно, время отклика будет очень большим (пропорционально сложности обработки). В итоге сервер будет не успевать обрабатывать все запросы. Кроме того, если сервер работает на двухъядерном узле, то мы будем наблюдать полную загрузку одного ядра и простой второго.

Именно поэтому в реальных приложениях этот подход используется крайне редко. Мы далее не будем касаться его.

5.2.2.2. Обработка каждого нового запроса в отдельном потоке

При такой схеме для каждого клиентского запроса создается отдельный поток. В рассматриваемом нами приложении реализован именно этот подход. Сервер имеет следующую архитектуру: основной поток приложения ожидает поступления запросов от клиентов и при поступлении нового создает поток, передавая клиентский запрос ему на обработку. Созданный поток выполняет соответствующую обработку и завершает свое существование.

Однако и этот подход имеет существенные недостатки:

- Частое создание и завершение потоков. Создание и завершение потока – весьма трудоемкая операция, требующая времени и ресурсов, поэтому уровень непроизводительных издержек очень высок.
- Нерегулируемое количество потоков. Это может привести к избыточному потреблению оперативной памяти, а также к резкому уменьшению свободной части виртуального адресного пространства процесса.
- Большое количество переключений контекстов рабочих потоков.

5.2.2.3. Организация пула потоков

Пул потоков предлагает решение перечисленных выше проблем. Стандартная схема организации приложения с пулом потоков выглядит следующим образом. Имеются основной поток приложения, ожидающий поступления клиентских запросов, и потоки, составляющие пул, которые создаются заранее или при поступлении первого запроса. При поступлении запроса главный поток выбирает свободный поток из пула и передает запрос ему на обработку, если же свободных потоков нет, то запрос помещается в очередь и ждет освобождения одного из потоков в пуле.

Положительный момент состоит в том, что потоки, однажды созданные, используются многократно, в результате чего издержки на создание и уничтожение потока малы по сравнению с его полезной работой. В итоге сокращается время обработки одного запроса, поскольку поток уже существует, когда прибывает очередной запрос.

Обычно имеет смысл ограничивать общее число рабочих потоков либо числом доступных процессоров (ядер), либо числом кратным ему (чтобы обеспечить

равномерную загрузку ядер). Если потоки занимаются только вычислениями, их число обычно выбирается равным числу ядер. Если же потоки некоторое время находятся в состоянии ожидания (выполнение операций ввода-вывода), то число потоков может превышать число ядер. Обычно ограничивают число потоков удвоенным числом процессоров (ядер).

5.2.3. Изменение архитектуры приложения: организация пула потоков

Посмотрим, какие изменения произойдут в работе нашего приложения при введении пула потоков. Для этого в **Microsoft Visual Studio** в окне **Solution Explorer** выберите проект **ClientServerPool** и дважды щелкните на файле **ClientServerPool.cpp**.

В начале файла **ClientServerPool.cpp** объявлена новая константа, указывающая количество потоков в пуле:

```
const int numPoolThreads = 4;
```

Кроме того, вводится еще одна очередь, используемая сервером для хранения запросов при отсутствии свободных потоков в пуле:

```
queue<int> serverRequests; //request queue on server
```

Ознакомьтесь с кодом приложения. После этого запустите процесс профилирования в ITP, в результате чего рабочая область ITP должна принять вид, как показано на рис. 4.

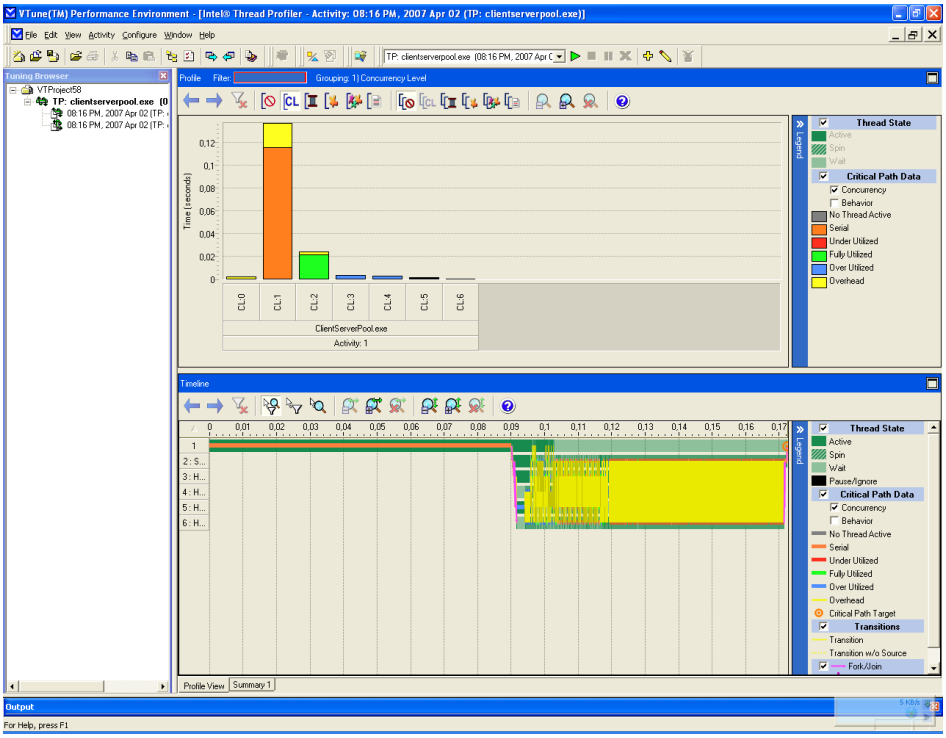

Рис. 4. Результаты профилирования приложения

При анализе можно заметить, что непроизводительные издержки значительно снизились, однако все еще присутствуют издержки, связанные с синхронизацией. Рассмотрим их подробнее в следующем разделе.

5.2.4. Выбор примитивов синхронизации

Основные издержки при синхронизации связаны с доступом потоков к глобальному массиву **requestsStatistics**, в котором хранится статистика запросов. При этом доступ происходит каждый раз в рамках критической области:

```
//enter to critical section (lock requestsStatisticsMutex mutex)
WaitForSingleObject(requestsStatisticsMutex,INFINITE);
//increment counter
requestsStatistics[serverRequest]++;
//release critical section (unlock requestsStatisticsMutex mutex)
ReleaseMutex(requestsStatisticsMutex);
```

Это сильно увеличивает время работы потоков-обработчиков, поскольку они вынуждены ожидать друг друга. Можно предложить следующие пути решения этой проблемы:

- использование объектов синхронизации пользовательского уровня (критическая секция), не использующих системные вызовы, вместо объектов уровня ядра операционной системы (мьютекс);
- предпочтение использования атомарных функций синхронизации ОС (**AtomicIncrement**, **AtomicDecrement** etc);
- использование локальных переменных вместо глобальных.

Рассмотрим второй из этих подходов. Закомментируйте критическую область для доступа к **requestsStatistics** и раскомментируйте следующую строку в рабочей функции потока обработчика:

```
InterlockedIncrement(reinterpret_cast<long*>(
  &requestsStatistics[serverRequest]));
```

Проанализируйте произошедшие изменения и время работы приложения с помощью ITP.

5.2.5. Снижение частоты синхронизации

Можно заметить, что в нашем приложении синхронизация происходит слишком часто (большое количество стрелок желтого цвета). Это вызвано тем, что потоки работают с одними и теми же глобальными объектами и вынуждены ждать друг друга. Эта проблема довольно часто встречается в многопоточных приложениях, и естественный подход к ее решению – снижение частоты синхронизации.

В нашем случае в качестве решения можно предложить следующий подход:

- на сервере вместо одной очереди сообщений создается **numTypes** очередей, в каждой из которых хранятся запросы одинакового типа;
- сервер для каждого пришедшего запроса определяет его тип и помещает его в соответствующую очередь сообщений;
- каждый поток обрабатывает сообщения только одного типа;

- каждый поток работает только со своим счетчиком, поэтому пропадает необходимость в синхронизации доступа к массиву **`requestsStatistics`**.

5.3. Контрольные вопросы

1. В чем преимущество создания пула потоков перед обработкой всех запросов в одном потоке и обработкой каждого запроса в новом потоке?
2. В чем недостатки использования примитивов синхронизации уровня ядра ОС?
3. Как еще можно увеличить эффективность приложения ClientServerPool?

5.4. Задания для самостоятельной работы

1. Реализуйте модель синхронизации с использованием критических секций, сравните ее по производительности с вариантами, использующими мьютексы и атомарные функции.
2. Реализуйте схему синхронизации, изложенную в пункте 5.2.5.

5.5. Литература

1. Developing Multithreaded Applications: A Platform Consistent Approach. Intel Corporation, March 2003.
2. Threading Methodology: Principles and Practices. Intel Corporation, March 2003.
3. Multi-Core Programming for Academia. Student Workbook, by Intel.
4. Akhter Sh., Roberts J. Multi-Core Programming Intel Press, 2006.
5. Intel® Thread Profiler. Getting Started Guide.
6. Intel® Thread Profiler. Guide to Sample Code.
7. Intel® Thread Profiler Help.
8. Intel® Thread Profiler. Краткое описание». Материалы по образовательному комплексу «Технологии разработки параллельных программ».
9. Ширшов А. Эффективная многопоточность // RSDN Magazine. 2003. #2.

Часть III. Создание параллельной программы

6. Библиотека Intel Threading Building Blocks. Краткое описание

Разработка программного обеспечения, всегда бывшая делом непростым, проходит в настоящий момент очередной виток повышения сложности, вызванный повсеместным распространением многоядерных процессоров, использовать все возможности которых можно лишь создавая многопоточные программы. С одной стороны, все необходимые для этого средства уже весьма давно существуют в распространенных операционных системах семейств Microsoft Windows и Unix/Linux, с другой – для программистов-прикладников использование этих механизмов по уровню удобства и объему необходимых знаний немногим легче, чем было когда-то программирование на ассемблере. Остро необходимы инструменты, берущие на себя по возможности большую часть задач, связанных с обслуживанием «параллельности» в многопоточных программах, и дающие разработчику возможность сосредоточиться на решении конкретных прикладных задач. В настоящем разделе представлено краткое описание одного из инструментов, пытающегося «играть на указанном поле», – библиотеки Intel® Threading Building Blocks [1].

Возможности библиотеки, рассмотренные в настоящем описании, подробно изучаются в лабораторных работах «Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения» и «Использование механизма логических задач библиотеки Intel Threading Building Blocks на примере вычисления быстрого преобразования Фурье».

6.1. Назначение библиотеки Intel Threading Building Blocks

Intel® Threading Building Blocks (TBB) – библиотека, предназначенная для разработки параллельных программ для систем с общей памятью. В отличие от других известных подходов и инструментов: программирования непосредственно в потоках, использования OpenMP – TBB и сама написана на языке C++ (в классах и шаблонах) и ее использование предполагает и дает возможность разработки параллельной программы в объектах. Кроме того, библиотека скрывает низкоуровневую работу с потоками, упрощая тем самым процесс создания параллельной программы.

6.2. Возможности библиотеки Intel Threading Building Blocks

В состав TBB входит набор классов и функций, позволяющих решать следующие типичные для разработки параллельных программ задачи:

- распараллеливание циклов с известным числом повторений;
- распараллеливание циклов с известным числом повторений с редукцией;
- распараллеливание циклов с условием;
- распараллеливание рекурсии.

Также библиотека содержит:

- потокобезопасные контейнеры (аналогичны контейнерам STL, за исключением того, что накладных расходов при работе с ними в параллельных приложениях меньше, чем при использовании стандартных контейнеров STL);
- операторы выделения динамической памяти (*аллокаторы*);
- примитивы синхронизации.

Полную информацию о составе и возможностях библиотеки TBB можно найти в [2, 3].

6.3. Описание библиотеки Intel Threading Building Blocks

TBB является межплатформенной библиотекой – на момент написания данного пособия существуют реализации под операционные системы Microsoft Windows, Linux, Mac OS. Кроме того, библиотека свободно распространяется в некоммерческих целях.

В отличие от OpenMP библиотека Intel® Threading Building Blocks не требует поддержки со стороны компилятора. Для сборки приложения необходима установленная версия TBB, а для запуска под операционной системой семейства Microsoft Windows достаточно иметь динамическую библиотеку **`tbb.dll`**. Подробные сведения о сборке и запуске программы, использующей TBB, приведены в приложении 6.11.2 «Сборка и настройка проекта».

6.4. Инициализация и завершение библиотеки

Для использования возможностей TBB по распараллеливанию вычислений необходимо иметь хотя бы один активный (инициализированный) экземпляр класса **`tbb::task_scheduler_init`**. Этот класс предназначен для создания потоков и внутренних структур, необходимых планировщику потоков для работы.

Объект класса **`tbb::task_scheduler_init`** может находиться в одном из двух состояний: активном или неактивном. Активировать экземпляр класса **`tbb::task_scheduler_init`** можно двумя способами:

- непосредственно при создании объекта **`tbb::task_scheduler_init`**. При этом число создаваемых потоков может определяться автоматически библиотекой или задаваться вручную пользователем;
- отложенной инициализацией при помощи вызова метода **`task_scheduler_init::initialize`**.

Прототип конструктора класса **`tbb::task_scheduler_init`** представлен ниже:

```
task_scheduler_init(int number_of_threads = automatic);
```

Доступны следующие варианты значений параметра **`number_of_threads`**:

- **`task_scheduler_init::automatic`** (как видно из прототипа, это значение по умолчанию). Библиотека автоматически определяет и создает оп-

тимальное количество потоков для данной вычислительной системы. В приложениях с большим числом компонентов определить оптимальное число потоков непросто, в этом случае можно положиться на планировщик потоков библиотеки, который определит их оптимальное число автоматически, поэтому значение **task_scheduler_init::automatic** рекомендуется использовать в release-версиях приложений. Пример инициализации объекта класса **tbb::task_scheduler_init** данным способом представлен ниже:

```
task_scheduler_init init; // Инициализация объекта класса
                         // tbb::task_scheduler_init
                         // по умолчанию при создании объекта
```

- Положительное число – число потоков, которое будет создано библиотекой. Потоки создаются сразу после вызова конструктора. Пример инициализации экземпляра класса **tbb::task_scheduler_init** данным способом представлен ниже:

```
task_scheduler_init init(3); // Инициализация объекта класса
                             // tbb::task_scheduler_init с 3
                             // потоками при создании объекта
```

- **task_scheduler_init::deferred** – отложенная инициализация объекта класса **tbb::task_scheduler_init**. Инициализация происходит только после вызова метода **task_scheduler_init::initialize**. Прототип метода **task_scheduler_init::initialize** представлен ниже:

```
void initialize(int number_of_threads = automatic);
```

Аргумент этого метода имеет те же варианты, что и аргумент конструктора класса **tbb::task_scheduler_init**. Пример инициализации объекта класса **tbb::task_scheduler_init** данным способом представлен ниже.

```
task_scheduler_init init(task_scheduler_init::deferred);
init.initialize(3);   // Инициализация объекта класса
                      // tbb::task_scheduler_init с 3 потоками
```

Перед завершением работы приложения необходимо перевести объект класса **tbb::task_scheduler_init** в неактивное состояние (завершить работу всех созданных потоков, уничтожить все созданные объекты). Это происходит автоматически в деструкторе класса **task_scheduler_init**. Также можно деактивировать объект класса **tbb::task_scheduler_init** в любой требуемый момент, чтобы освободить ресурсы системы под другие нужды. Для этих целей существует метод **task_scheduler_init::terminate**. Его прототип представлен ниже:

```
void terminate();
```

После вызова метода **task_scheduler_init::terminate** можно повторно активировать объект класса **tbb::task_scheduler_init**, вызвав метод **task_scheduler_init::initialize.**

Если в приложении уже активен один экземпляр класса **tbb::task_scheduler_init**, то при создании нового объекта его параметры (число потоков) будут проигнорированы. Поэтому для изменения числа потоков библиотеки нужно перевести текущий экземпляр класса **task_scheduler_init** в неактивное состояние, вызвав метод **task_scheduler_init::terminate,** или уничтожить объект, вызвав его деструктор. После этого можно вызывать метод **task_**

scheduler_init::initialize для нового объекта или создать объект класса **tbb::task_scheduler_init**, указав необходимое число потоков.

Рассмотренные операции требуют значительных временных ресурсов, поэтому типичная схема работы с объектом класса **tbb::task_scheduler_init** – инициализация в начале работы, деинициализация в конце работы приложения. Ниже приведен пример типичной структуры TBB-программы.

```
#include "tbb/task_scheduler_init.h"
// Подключение необходимых заголовочных файлов
using namespace tbb;

int main()
{
  task_scheduler_init init;
  // Вычисления
  return 0;
}
```

В примере для того чтобы создать экземпляр класса **tbb::task_scheduler_init**, был подключен заголовочный файл **task_scheduler_init.h**. Аналогичное происходит со всеми остальными функциями/классами библиотеки – воспользоваться ими можно, подключив соответствующий их названию заголовочный файл. Полный список функций/классов и соответствующих им заголовочных файлов представлен в приложении 6.11.1 «Заголовочные файлы библиотеки TBB».

Более сложный пример работы с библиотекой:

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;

int main()
{
  task_scheduler_init init; // Инициализация по умолчанию
  //Вычисления 1
  init.terminate();         // Деинициализация
  init.initialize(4);       // Инициализация с 4 потоками
  //Вычисления 2
  return 0;
}                       // Деинициализация при уничтожении (вызов
                        // деструктора) объекта init
```

Библиотека TBB может использоваться совместно с библиотекой OpenMP. Для этого на каждом потоке, созданном с помощью OpenMP (внутри параллельной секции), необходимо создать активный объект класса **tbb::task_scheduler_init** библиотеки TBB. Более подробная информация об этом представлена в приложении 6.11.3 «Совместное использование с OpenMP».

6.5. Распараллеливание простых циклов

6.5.1. Циклы с известным числом повторений

Вычисления с заранее определенным числом итераций обычно происходят с использованием цикла **for**. Библиотека TBB дает возможность реализовать парал-

лельную версию таких вычислений. Для этого библиотека предоставляет шаблонную функцию **tbb::parallel_for.** Прототип этой функции представлен ниже:

```
template<typename Range, typename Body>
void parallel_for(const Range& range, const Body& body);
```

Как видно из прототипа, функция **tbb::parallel_for** имеет два шаблонных параметра. Первый параметр представляет итерационное пространство – класс специального вида, задающий количество итераций цикла. Второй параметр – функтор[6], класс, реализующий вычисления цикла через метод **body::operator().**

Итерационное пространство

Первый аргумент функции **tbb::parallel_for** – итерационное пространство.

Библиотека TBB содержит два реализованных итерационных пространства: одномерное итерационное пространство **tbb::blocked_range** и двумерное итерационное пространство **tbb::blocked_range2d**. Пользователь библиотеки может реализовать и свои итерационные пространства.

Одномерное итерационное пространство **tbb::blocked_range** задает диапазон в виде полуинтервала **[begin, end)**, где тип элементов **begin** и **end** задается через шаблон. В качестве параметра шаблона могут быть использованы: тип **int**, указатели, STL-итераторы прямого доступа и др. (список требований, предъявляемых к шаблону, представлен в [3]).

Класс **tbb::blocked_range** имеет три основных поля: **my_begin**, **my_end**, **my_grainsize.** Эти поля расположены в секции **private**, получить их значения можно только с помощью методов: **begin**, **end**, **grainsize**. Поля **my_begin** и **my_end** задают левую и правую границы полуинтервала **[my_begin, my_end).** Поле **my_grainsize** имеет целый тип и задает размер порции вычислений. Прототип основного конструктора класса **blocked_range** представлен ниже:

```
blocked_range::blocked_range(Value begin, Value end,
  size_t grainsize = 1);
```

Основной операцией любого итерационного пространства является его *расщепление*. Операция расщепления выполняется с использованием *конструктора расщепления*, задача которого – разделить итерационное пространство на два подмножества. Для **tbb::blocked_range** разделение итерационного пространства выполняется на два подмножества равного (с точностью до округления) размера. Например, итерационное пространство **a**, созданное представленным ниже способом, задает полуинтервал [5, 14) и размер порции вычислений, равный 2. Итерационное пространство **b** создается с помощью конструктора расщепления на основе итерационного пространства **a.**

```
blocked_range<int> a(5, 14, 2);
blocked_range<int> b(a, split());
```

[6] В C++ *функторами* или *функциональными классами* называют классы специального вида, основная функциональность которых сосредоточена в методе operator(). Функторы активно используются во многих библиотеках классов, и TBB не является исключением.

После вызова конструктора расщепления будет создан еще один объект того же типа и будут пересчитаны диапазоны как в новом, так и в старом объекте (рис. 1). Значение поля **my_grainsize** не изменится.

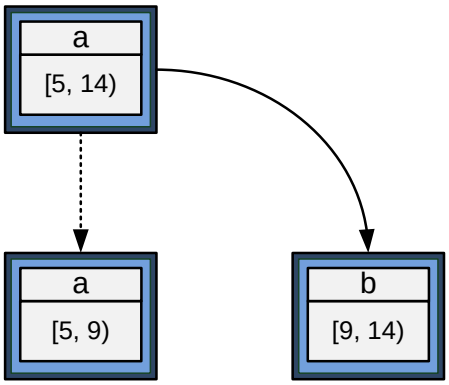

Рис. 1. Расщепление итерационного пространства blocked_range(5,14)

Рассмотрим пример использования одномерного итерационного пространства:

```
blocked_range<int> range(0, 100);
for (int i = range.begin(); i != range.end(); i++)
{
  //Вычисления
}
```

Представленный пример является полным аналогом следующего:

```
for (int i = 0; i != 100; i++)
{
  //Вычисления
}
```

Двумерное итерационное пространство **tbb::blocked_range2d** является полным аналогом одномерного, за исключением того, что оно задает двумерный полуинтервал вида **[x1, x2)×[y1, y2).**

Пользователю библиотеки TBB предоставляется возможность создать свое итерационное пространство. Для этого необходимо определить класс, в котором следует реализовать методы:

- **Range(const R&)** – конструктор копирования.
- **~Range()** – деструктор.
- **bool empty()** – метод проверки итерационного пространства на пустоту. Если оно пусто, то функция должна вернуть **true.**
- **bool is_divisible()** – метод проверки на возможность разделения итерационного пространства. Если разделение возможно, то функция должна вернуть **true.**
- **Range(R& r, split)** – конструктор расщепления, создает копию итерационного пространства и разделяет диапазон, задаваемый итерационным пространством, на две части (изменяется диапазон итерационного пространства как вновь созданного объекта, так и объекта, его породившего).

Параметр **split** (служебный класс без полей и методов) предназначен для того, чтобы отличить конструктор копирования от конструктора расщепления. Его реализация представлена ниже:

```
namespace tbb
{
  class split
  {
  };
}
```

Пример реализации простого одномерного итерационного пространства представлен ниже. Конструктор копирования и деструктор явно не реализованы, т.к. их реализация по умолчанию является корректной.

```
class SimpleRange
{
private:
  int my_begin;
  int my_end;

public:
  int begin() const { return my_begin; }
  int end() const { return my_end; }

  bool empty() const { return my_begin == my_end; }
  bool is_divisible() const { return my_end > my_begin + 1; }

  SimpleRange(int begin, int end): my_begin(begin), my_end(end)
  {}

  SimpleRange(SimpleRange& r, split )
  {
    int medium = (r.my_begin + r.my_end) / 2;
    my_begin = medium;
    my_end = r.my_end;
    r.my_end = medium;
  }
};
```

Заметим, что в классе **SimpleRange** мы не объявили поле **my_grainsize**, которое задавало размер порции вычислений в классе **tbb::blocked_range**. Его наличие в общем случае не является обязательным, т.к. размер порции вычислений на самом деле определяется реализацией метода **is_divisible**. В указанной реализации этот размер равен 1. Более подробно о порядке обработки итерационного пространства см. пункт «Планирование вычислений».

Функтор

Второй аргумент функции **tbb::parallel_for** – функтор. Функтор – это класс специального вида, который выполняет необходимые вычисления с помощью метода **operator()**. В первом приближении можно считать, что функтор получается в результате трансформации тела цикла в класс.

Функтор для функции **tbb::parallel_for** должен содержать следующие методы:

- конструктор копирования, необходимый для корректной работы функции **tbb::parallel_for**, которая создает копии функтора в соответствии с принятым разработчиками библиотеки алгоритмом реализации параллелизма;
- деструктор **~Body()**;
- метод **operator()**, выполняющий вычисления.

Аргументом последнего метода является итерационное пространство. Прототип метода представлен ниже:

```
void operator()(Range& range) const
```

Метод **operator()** является основным в функторе. Метод объявлен константным, поскольку не нуждается в изменении значений полей функтора, если таковые в нем имеются. Ниже мы убедимся в этом факте на конкретном примере.

Одним из примеров «хорошего» параллелизма является задача, в которой итерации цикла могут выполняться без взаимной синхронизации. В качестве таковой рассмотрим задачу умножения матрицы на вектор. В данной задаче:

- операции скалярного умножения векторов, на которых основаны вычисления, выполняются без синхронизаций;
- запись результатов умножения происходит ровно один раз для каждого скалярного умножения векторов, в дальнейших вычислениях уже посчитанные результаты не используются;
- данные для вычислений (элементы массива и вектора) не изменяются во время вычислений.

Функтор является функциональным классом и не должен содержать в себе ни обрабатываемые данные, ни получаемый результат. Именно поэтому все поля представленного функтора, умножающего матрицу на вектор, являются указателями на внешние данные, кроме числа столбцов матрицы. Для последнего поля сделано исключение только потому, что на его хранение в виде копии требуется столько же памяти (размер типа **int**), сколько и на хранение указателя. Инициализация полей осуществляется с помощью конструктора. Как мы увидим далее, в процессе работы функции **tbb::parallel_for** на основе первоначально переданного в нее функтора создается некоторое количество копий. Естественно, все они благодаря тому, что поля-указатели адресуют одни и те же данные, будут «разделять» и исходные данные и результат, что, собственно говоря, нам и нужно.

```
//Скалярное умножение векторов
double VectorsMultiplication(const double *a, const double *b,
  int size)
{
  double result = 0.0;
  for(int i = 0; i < size; i++)
    result += a[i] * b[i];
  return result;
}

//Функтор
class VectorsMultiplicator
{
  const double *matrix, *vector; // Исходные данные для умножения
```

```
  double *const resultVector;    // Вектор результатов
  int const numOfColumns;        // Количество столбцов матрицы
public:
  VectorsMultiplicator(double *tmatrix, double *tvector,
    double *tresultVector, int tnumOfColumns) :
    matrix(tmatrix), vector(tvector),
    resultVector(tresultVector), numOfColumns(tnumOfColumns)
  {}

  void operator()(const blocked_range<int>& r) const
  {
    int begin = r.begin(),  end = r.end();

    for (int i = begin; i != end; i++)
      resultVector[i] = VectorsMultiplication(
        &(matrix[i * numOfColumns]), vector, numOfColumns);
  }
};
```

Заметим, что в классе **VectorsMultiplicator** не реализован ни конструктор копирования, ни деструктор, несмотря на наличие полей-указателей. Над объяснением этого факта предлагаем читателю подумать самостоятельно.

Планирование вычислений

Алгоритм работы функции **tbb::parallel_for** устроен таким образом, что планирование вычислений осуществляется динамически, то есть на этапе выполнения. Определяющим моментом планирования является то, как реализовано итерационное пространство. Рассмотрим алгоритм работы функции **tbb::parallel_for** при использовании одномерного итерационного пространства.

Одним из полей одномерного итерационного пространства является размер порции вычислений – **grainsize**. Его значение является определяющим при планировании вычислений. Функция **tbb::parallel_for** распределяет на выполнение между всеми потоками части итерационного пространства размером **grainsize**. Если **grainsize** равно размеру итерационного пространства (общему числу итераций), то все итерации будут выполнены на одном потоке. Если **grainsize** равно <общее число итераций>/<число потоков>, то каждый поток, скорее всего (т.к. планирование осуществляется динамически, то точно сказать нельзя), выполнит одинаковое число итераций, равное **grainsize**. Если **grainsize** меньше, чем <общее число итераций>/<число потоков>, то планировщик потоков распределит итерации между потоками по специальному алгоритму.

Значение **grainsize** выбирается разработчиком приложения. При этом малое значение **grainsize** способствует увеличению масштабируемости приложения (запуск на системе с большим количеством процессоров/ядер приведет к большему ускорению). Например, если значение **grainsize** равно половине итерационного пространства, то при запуске на машине с 4 процессорами работа будет выполняться только двумя из них, т.к. остальным ее просто не достанется из-за большого значения **grainsize**. Поэтому необходимо устанавливать маленькие значения **grainsize** для большей масштабируемости приложения. Работа планировщика потоков занимает определенное время, поэтому чем меньше значение **grainsize**, тем больше времени потребуется функции **tbb::parallel_for** на

распределение заданий. Таким образом, при очень малых значениях **grainsize** приложение будет обладать очень хорошей масштабируемостью, но при этом будет работать очень неэффективно из-за больших накладных расходов на работу планировщика. При очень больших значениях **grainsize** приложение будет работать максимально эффективно, но его масштабируемость будет очень плоха.

Итог:

- параметр **grainsize** не должен быть слишком маленьким, т.к. это может негативно отразиться на времени работы приложения (большие накладные расходы на работу функции **tbb::parallel_for**);
- параметр **grainsize** не должен быть слишком большим, т.к. это может негативно отразиться на масштабируемости приложения.

Большинство вычислительных систем очень сложны, чтобы можно было теоретически подобрать оптимальное значение **grainsize**, поэтому рекомендуется подбирать его экспериментально.

Алгоритм экспериментального подбора значения **grainsize** следующий [3]:

1. Установите значение **grainsize** достаточно большим, например равным размеру итерационного пространства в случае использования класса **blocked_range.**
2. Запустите приложение в один поток, измерьте время его выполнения.
3. Установите значение **grainsize** в 2 раза меньше, запустите приложение по-прежнему в один поток и оцените замедление по отношению к шагу 2. Если приложение замедлилось на 5–10%, это хороший результат. Продолжайте уменьшение **grainsize** до тех пор, пока замедление не превысит 5–10%.

Рассмотрим алгоритм работы **tbb::parallel_for** с точки зрения распределения вычислений на следующем примере:

```
parallel_for(blocked_range<int>(5, 14, 2), body);
```

На первом шаге имеется функтор **body** и одномерное итерационное пространство, размер которого равен 14 – 5 = 9. Это значение больше, чем размер порции, равный 2, поэтому функция **tbb::parallel_for** расщепляет итерационное пространство на два, одновременно создавая для нового итерационного пространства собственный функтор (через конструктор копирования) и меняя, как мы уже отмечали выше, размер итерационного пространства для старого функтора. Данный процесс будет происходить рекурсивно до тех пор, пока размер очередного итерационного пространства будет не больше 2. После этого для каждого созданного функтора будет вызван метод **body::operator()** с сопоставленным с этим функтором итерационным пространством в качестве параметра. Алгоритм работы **tbb::parallel_for** представлен на рис. 2. Надпись **new** над стрелкой означает, что создается новый экземпляр итерационного пространства и функтора. Пунктирная стрелка означает, что изменяется диапазон, задаваемый итерационным пространством, при этом создание новых экземпляров функтора и итерационного пространства не происходит.

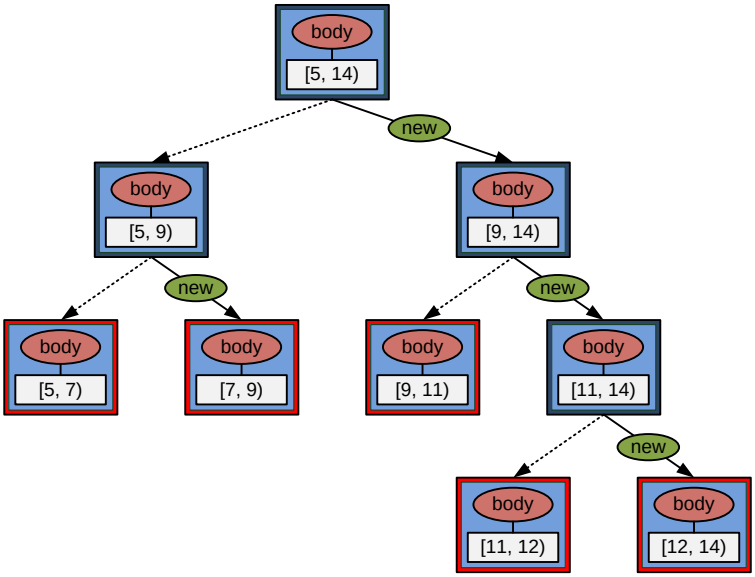

Рис. 2. Алгоритм работы функции **`tbb::parallel_for`**

Как видно из рисунка, после первого шага будут существовать два непересекающихся итерационных пространства. Если в библиотеке создано больше одного потока, то дальнейшая обработка полученных поддеревьев будет происходить параллельно. При этом процесс вычисления является недетерминированным и может выполняться как показано на рис. 3.

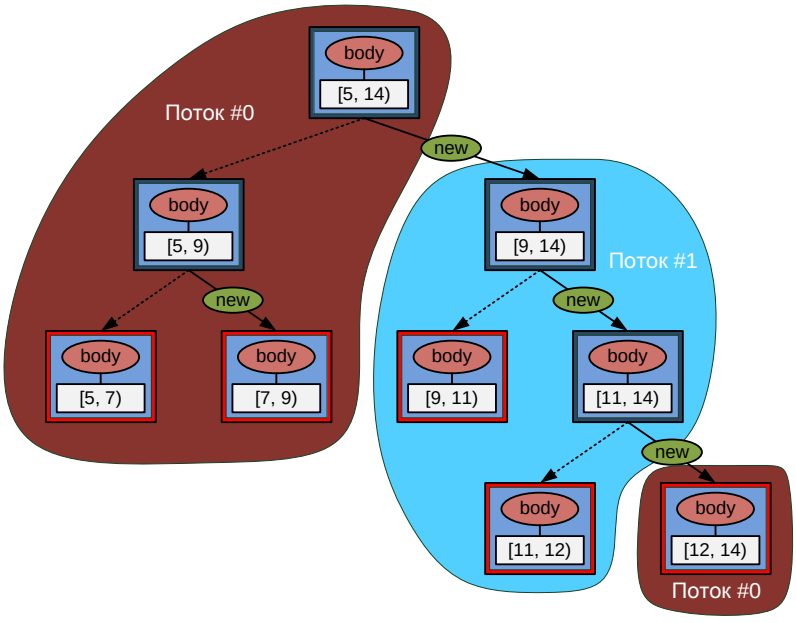

Рис. 3. Пример работы функции **`tbb::parallel_for`** при использовании двух потоков

Пример использования **tbb::parallel_for** в задаче умножения матрицы на вектор приведен ниже (**numOfRows** – количество строк матрицы, **numOfColumns** – количество столбцов матрицы):

```
parallel_for(blocked_range<int>(0, numOfRows, grainsize),
  VectorsMultiplicator(matrix, vector,
    resultVector, numOfColumns));
```

Для лучшего понимания функции **tbb::parallel_for** приведем практически аналогичный по устройству вариант распараллеливания задачи умножения матрицы на вектор на OpenMP:

```
#pragma omp parallel for schedule(dynamic, grainsize)
for(int i = 0; i < numOfRows; i++)
  resultVector[i] =
    VectorsMultiplication(&(matrix[i*columns]),
      vector, numOfColumns);
```

Представленное сравнение явно выглядит не в пользу TBB, во всяком случае по затратам на разработку. Однако задача, которую мы взяли здесь в качестве примера, довольно проста. В более сложных ситуациях возможности TBB явным образом задавать алгоритм планирования вычислений, инкапсулировать связанные с вычислительным потоком код и данные способствуют упрощению разработки и программного кода приложения.

6.5.2. Циклы с известным числом повторений с редукцией

Функция **tbb::parallel_reduce** предназначена для распараллеливания вычислений, представленных в виде цикла **for** с редукцией (типичным примером задачи, в параллельной реализации которой выгодно использовать редукцию, является скалярное умножение векторов).

Прототип функции **tbb::parallel_reduce** представлен ниже:

```
template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body);
```

Нетрудно видеть, что прототип **tbb::parallel_reduce** полностью совпадает с прототипом функции **tbb::parallel_for**, однако методы функторов для этих функций различны. Функтор **body** для **tbb::parallel_reduce** должен содержать следующие методы:

- Конструктор расщепления **Body(body&, split)**.
- Деструктор **~Body()**;
- Метод, выполняющий вычисления **void operator()(Range& range)**. Аргументом является итерационное пространство. Заметим, что в отличие от функтора функции **tbb::parallel_for** у функтора функции **tbb::parallel_reduce** вычислительный метод не является константным, а это означает, что в методе можно изменять поля класса. Данное требование связано с необходимостью сохранения промежуточных результатов, которые будут использоваться при выполнении операции редукции для получения окончательного результата.
- Метод, выполняющий редукцию **void join(Body& rhs)**. В качестве параметра принимает ссылку на функтор, который выполнил часть вычислений. Посчитанные им данные должны быть учтены текущим функ-

тором (**this**), для получения окончательного результата. Функтор, переданный по ссылке, автоматически уничтожается после завершения редукции (вызова функции **join**).

Алгоритм работы **tbb::parallel_reduce** похож на **tbb::parallel_for**. В отличие от **tbb::parallel_for** функция **tbb::parallel_reduce** выполняет дополнительный этап вычислений – редукцию и, в зависимости от того, как происходит распределение вычислений по потокам, реализует одну из двух схем. Алгоритм работы **tbb::parallel_reduce** с точки зрения распределения вычислений рассмотрим на примере:

```
parallel_reduce(blocked_range<int>(5, 14, 2), body);
```

Прежде всего, отметим, что **tbb::parallel_reduce**, так же как и **tbb::parallel_for**, расщепляет итерационные пространства до тех пор, пока их размеры больше, чем **grainsize**. Однако отличие в работе функции **tbb::parallel_reduce** состоит в том, что она не создает копии исходного функтора при каждом расщеплении (кроме отдельного описанного ниже случая), а оперирует ссылками на функторы. Наконец, еще раз подчеркнем, что создание «порции» вычислений в виде итерационного пространства и связанного с ним функтора может выполняться и часто выполняется отдельно от дальнейшей обработки этого «порции».

Итак, первая схема работы **tbb::parallel_reduce** реализуется в том случае, если очередная «порция» вычислений обрабатывается на том же потоке, что и предыдущая. В данной схеме требуется и существует только один экземпляр функтора. Пусть, например, на очередной итерации потоком 0 была создана «порция» вычислений размером [9, 14). Пусть этот же поток обрабатывает эту «порцию» (рис. 4). Так как размер итерационного пространства больше 2, происходит его расщепление и копирование указателя на функтор. Расщепление итерационного пространства осуществляется так же, как и при использовании **tbb::parallel_for**. Данная схема реализуется всегда при вычислении в один поток.

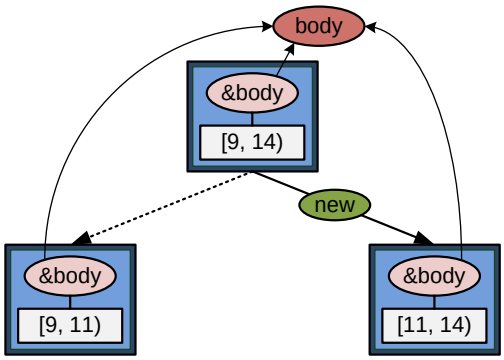

Рис. 4. Выполнение вычислений **tbb::parallel_reduce** на одном потоке

В том случае если очередная «порция» вычислений выполняется на потоке, отличном от потока – создателя «порции», то реализуется вторая схема путем создания нового функтора с помощью конструктора расщепления, как показано на рис. 5. Отметим, что, в отличие от итерационных пространств, конструктор рас-

щепления функтора реально никакого «разделения» функтора или его полей на части не производит. В большинстве случаев его работа совпадает с работой конструктора копирования. Использование конструктора расщепления в данном случае лишь дает возможность разработчикам функтора реализовать более сложное поведение в момент «переноса» функтора на другой поток.

Например, пусть на очередной итерации потоком 0 были созданы две «порции» вычислений [5, 9) и [9, 14). Далее поток 0 продолжил выполнять вычисления с «порцией» [5, 9) (на рис. 5 эта часть не показана), а поток 1 взял на выполнение «порцию» [9, 14). Во избежание возможных гонок данных поток 1 не использует ссылку на существующий функтор **`body1`**, а создает новый функтор **`body2`** с помощью конструктора расщепления и продолжает работать с ним, подставляя ссылку на него в «порцию» вместо исходного функтора **`body1`**. Далее все происходит так же, как в первой схеме.

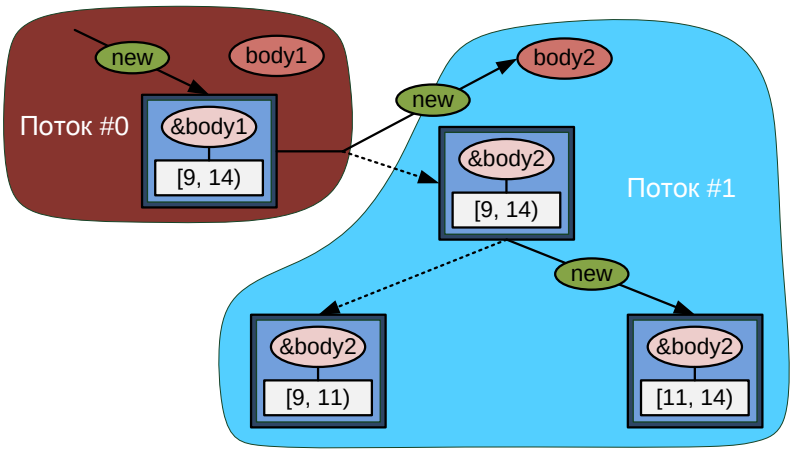

Рис. 5. Выполнение вычислений **`tbb::parallel_reduce`** при «смене» потока

Мы рассмотрели порядок выполнения «прямого хода» вычислений в цикле с редукцией. Теперь рассмотрим само выполнение операции редукции. Если все вычисления происходили в один поток, то операция редукции не требуется, т.к. функтор существует в одном экземпляре и он один выполнил все вычисления. Если же в некоторый момент очередная «порция» вычислений была создана одним потоком, а сами вычисления производились другим, то был создан новый функтор, а это значит, что необходимо выполнить редукцию нового функтора на старый.

Рассмотрим алгоритм редукции в функции **`tbb::parallel_reduce`** с точки зрения выполнения вычислений на примере:

```
parallel_reduce(blocked_range<int>(5, 14, 2), body);
```

Как и для **`tbb::parallel_for`**, процесс вычислений является недетерминированным (какой поток выполнит конкретную часть вычислений, определяется только на этапе выполнения). Пусть реализовалась ситуация, представленная на рис. 6.

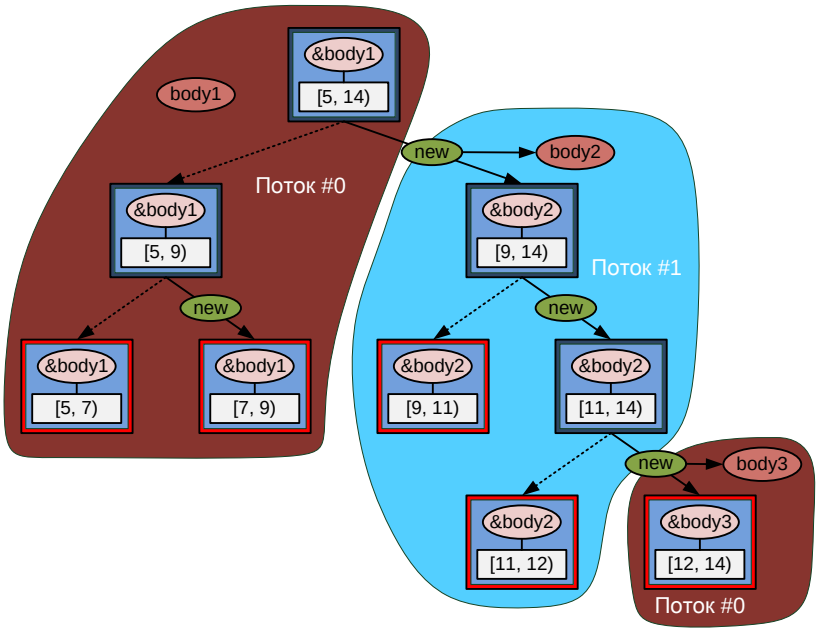

Рис. 6. Алгоритм работы **`tbb::parallel_reduce`**

В тех узлах дерева, где происходило расщепление функтора, по завершении вычислений будут выполнены операции редукции, как показано на рис. 7.

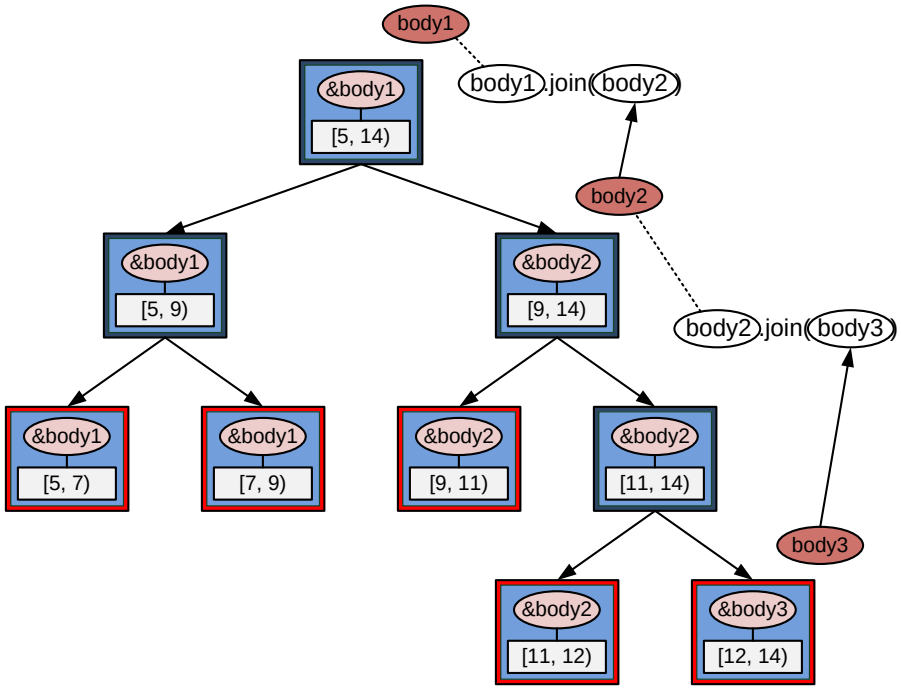

Рис. 7. Выполнение редукции при использовании **`tbb::parallel_reduce`**

Пример функтора функции **tbb::parallel_reduce** для решения задачи скалярного умножения векторов:

```
//Скалярное умножение векторов
double VectorsMultiplication(double *v1, double *v2, int size)
{
  double result = 0;
  for (int i = 0; i < size; i++)
    result += v1[i] * v2[i];
  return result;
}

//Функтор
class ScalarMultiplicator
{
private:
  const double *a, *b;
  double c;

public:
  explicit ScalarMultiplicator(double *ta, double *tb):
    a(ta), b(tb), c(0)
  {}

  ScalarMultiplicator(const ScalarMultiplicator& m, split):
    a(m.a), b(m.b), c(0)
  {}

  void operator()(const blocked_range<int>& r)
  {
    int begin = r.begin(),
        end = r.end();
    c += VectorsMultiplication(&(a[begin]), &(b[begin]),
         end - begin);
  }

  void join(const ScalarMultiplicator& multiplicator)
  {
    c += multiplicator.c;
  }

  double Result()
  {
    return c;
  }
};
```

Пример использования **tbb::parallel_reduce** в задаче скалярного умножения векторов приведен ниже (**size** – размер векторов; **a**, **b** – исходные векторы):

```
  ScalarMultiplicator s(a, b);
  parallel_reduce(blocked_range<int>(0, size, grainsize), s);
```

Для лучшего понимания функции **tbb::parallel_reduce** приведем практически аналогичный по устройству вариант распараллеливания задачи скалярного умножения векторов на OpenMP:

```
#pragma omp parallel for schedule(dynamic, grainsize) reduce(+: c)
for (int i = 0; i < size / grainsize; i++)
  c += VectorsMultiplication(&(a[i * grainsize]),
    &(b[i * grainsize]), grainsize);
```

6.6. Распараллеливание сложных конструкций

6.6.1. Сортировка

Библиотека TBB содержит шаблонную функцию **tbb::parallel_sort**, предназначенную для сортировки последовательности. С помощью этой функции можно выполнять параллельную сортировку встроенных типов языка C++ и всех классов, у которых реализованы методы **swap** и **operator()**. Последний должен выполнять сравнение двух элементов. Более подробная информация об этой функции представлена в [2, 3] .

Пример использования функции **tbb::parallel_sort** представлен ниже:

```
#include "tbb/parallel_sort.h"
#include <math.h>
using namespace tbb;
const int N = 100000;
float a[N];
float b[N];

void SortExample()
{
  for (int i = 0; i < N; i++)
  {
    a[i] = sin((double)i);
    b[i] = cos((double)i);
  }
  parallel_sort(a, a + N);
  parallel_sort(b, b + N, std::greater<float>());
}
```

6.6.2. Циклы с условием

Библиотека TBB содержит шаблонный класс **tbb::parallel_while**, с помощью которого можно выполнить параллельную обработку элементов, размещенных в некотором «входном» потоке данных. Элементы могут быть добавлены в поток данных во время вычислений.

Обозначим тип обрабатываемых элементов **element_type**, поток данных **S**, а класс, который будет заниматься обработкой элементов, **B**.

Класс обрабатываемых элементов должен содержать следующие обязательные методы:

- Конструктор по умолчанию **element_type()**.
- Конструктор копирования **element_type(const element_type&)**.
- Деструктор **~element_type()**.

Поток данных – класс, который должен содержать метод, возвращающий очередной элемент. Данный метод может выполняться только одним потоком в каждый момент времени. Прототип метода представлен ниже:

```
bool S::pop_if_present(element_type& item)
```

Класс, который будет заниматься обработкой элементов, должен содержать метод обработки элемента **B::operator()(element_type& item) const**. Этот метод может вызываться несколькими потоками одновременно и выполняться параллельно для разных элементов из потока данных. Также класс **B** должен содержать следующее переопределение типа:

```
typedef element_type argument_type;
```

6.6.3. Конвейерные вычисления

Библиотека TBB содержит класс **tbb::pipeline**, с помощью которого можно выполнять конвейерные вычисления. Для таких вычислений характерно выполнение нескольких стадий вычислений над одним и тем же элементом. Если хотя бы на одной из стадий работа над разными элементами может быть выполнена параллельно, то с помощью данного класса можно организовать такие вычисления.

Класс **tbb::pipeline** выполняет обработку элементов, заданных с помощью потока данных. Обработка осуществляется с помощью набора фильтров, которые необходимо применить к каждому элементу. Фильтры могут быть последовательными и параллельными. Для добавления фильтра в объект класса **tbb::pipeline** используется метод **add_filter**. Для запуска вычислений используется метод **run.**

Класс фильтра **tbb::filter** является абстрактным, должен быть унаследован всеми фильтрами, реализованными пользователем. Класс фильтра должен содержать следующие методы:

- **bool filter::is_serial() const** – возвращает тип фильтра: **true** – последовательный, **false** – параллельный;
- **virtual void* filter::operator()(void* item)** – метод обработки элементов. Должен вернуть указатель на элемент, который будет обрабатываться следующим фильтром. Первый фильтр, использующийся в объекте класса **tbb::pipeline,** должен вернуть **NULL**, если больше нет элементов для обработки;
- **virtual filter::~filter()** – деструктор.

Приведенное описание классов **tbb::pipeline** и **tbb::filter** является обзорным. Более подробная информация о реализации конвейерных вычислений представлена в [2].

6.7. Ядро библиотеки

Кроме набора высокоуровневых алгоритмов, которые предназначены для упрощения разработки параллельных программ, библиотека TBB предоставляет возможность писать параллельные программы на низком уровне – уровне «логических задач», работа с которыми, тем не менее, более удобна, чем напрямую с потоками.

6.7.1. Общая характеристика логических задач

Логическая задача в библиотеке TBB представлена в виде класса **tbb::task**. Этот класс является базовым при реализации задач, т.е. должен быть унаследован всеми пользовательскими логическими задачами. В дальнейшем под

логической задачей будем понимать любой класс, который является потомком класса **tbb::task**.

Класс **tbb::task** содержит виртуальный метод **task::execute**, в котором выполняются вычисления. Прототип этого метода представлен ниже:

```
task* task::execute()
```

В этом методе производятся необходимые вычисления, после чего возвращается указатель на следующую задачу, которую необходимо выполнить. Если возвращается **NULL**, то из пула готовых к выполнению задач выбирается новая.

Библиотека TBB содержит специально реализованную «пустую» задачу (**tbb::empty_task**), которая часто оказывается полезной. Метод **task::execute** этой задачи не выполняет никаких вычислений. Для наглядности приведем программный код этой задачи:

```
class empty_task: public task
{
  task* execute()
  {
    return NULL;
  }
};
```

6.7.2. Алгоритм работы

Рассмотрим общую идею работы планировщика потоков библиотеки TBB. Каждый поток, созданный библиотекой, имеет свое множество (пул) готовых к выполнению задач. Это множество представляет собой динамический массив списков. Списки обрабатываются в порядке LIFO (last-in first-out). Задачи, расположенные на i-м уровне (в списке i-го элемента массива), порождают подзадачи уровня $i+1$. Поток выполняет задачи из самого нижнего непустого списка из массива списков. Если все списки потока пусты, то поток случайным образом «забирает» задачи, расположенные в наиболее низком списке у других потоков. Пример работы планировщика потоков библиотеки TBB при запуске в 2 потока представлен на рис. 8.

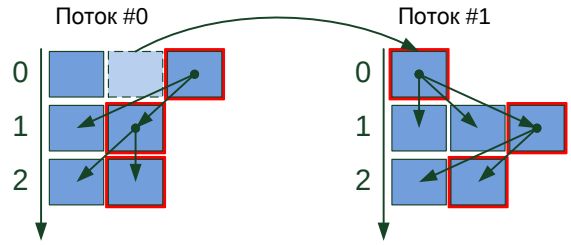

Рис. 8. Алгоритм работы планировщика потоков библиотеки TBB

Каждая задача имеет следующий набор связанных с ней атрибутов:

- **owner** – поток, которому принадлежит задача;

- **parent**[7] – равен либо **NULL**, либо указателю на другую задачу, у которой поле **refcount** будет уменьшено на единицу после завершения текущей задачи. Для получения значения этого атрибута предназначен метод **parent**;
- **depth** – глубина задачи в дереве задач. Получить значение этого атрибута можно с помощью метода **depth**, а установить с помощью **set_depth**;
- **refcount** – число задач, у которых текущая задача указана в поле **parent**. Получить значение поля **refcount** можно с помощью метода **ref_count**, а установить с помощью **set_ref_count**.

В дальнейшем каждую задачу будем характеризовать следующей тройкой: (**parent**, **depth**, **refcount**).

После того как планировщик потоков назначает каждому потоку задачу на выполнение, происходит следующее:

1. Выполнение метода **task::execute** и ожидание его завершения.
2. Если для задачи не был вызван один из методов вида **task::recycle_*** (**recycle_as_child_of**, **recycle_to_reexecute**, **recycle_as_continuation**, **recycle_as_safe_continuation**), то:
   - Если поле **parent** не **NULL**, то поле **parent->refcount** уменьшается на единицу. Если поле **parent->refcount** становится равным 0, то задача **parent** помещается в пул готовых к выполнению.
   - Вызов деструктора задачи.
   - Освобождение памяти, занимаемой задачей.
3. Если для задачи был вызван один из методов **task::recycle_***, то задача повторно добавляется в пул готовых к выполнению (отличия между этими методами будет представлено ниже).

6.7.3. Создание и уничтожение логических задач

Создание задачи должно осуществляться только с помощью оператора **new**, перегруженного в библиотеке TBB. Все виды оператора **new** представлены ниже:

- **new(task::allocate_root()) T** – выполняет создание «главной» задачи типа **T** со следующими атрибутами **(NULL, depth, 0)**. Для запуска этого типа задач необходимо использовать метод **task::spawn_root_and_wait**;
- **new(this.allocate_child()) T** – выполняет создание подчиненной задачи типа **T** для задачи **this** со следующими атрибутами **(this, depth + 1, 0)**. Атрибуты задачи **this (parent, depth, refcount)** автоматически изменяются на **(parent, depth, refcount + 1)**;
- **new(this.allocate_continuation()) T** – выполняет создание задачи того же уровня, что и задача **this**. Атрибуты задачи **this (parent, depth, refcount)** автоматически изменяются на **(NULL, depth, refcount)**, новая задача создается со следующими атрибутами **(parent, depth, 0)**;

[7] Здесь и далее обозначения parent, depth используются и как имена полей, и как имена методов, с помощью которых можно получить их значения.

- **new(this.task::allocate_additional_child_of(parent))** – выполняет создание подчиненной задачи для произвольной задачи, указанной в качестве параметра. Атрибуты задачи **parent (grandparent, depth, refcount)** автоматически изменяются на **(grandparent, depth, refcount + 1)**, новая задача создается со следующими атрибутами **(parent, depth + 1, 0)**.

Пример создания задачи представлен ниже:

```
task* MyTask::execute()
{
  // ...
  MyTask &t = *new (allocate_child()) MyTask();
  // ...
}
```

Уничтожение задачи осуществляется автоматически с помощью виртуального деструктора. Также можно уничтожить задачу вручную с помощью метода **task::destroy**. При этом поле **refcount** уничтожаемой задачи должно быть равно 0. Прототип метода представлен ниже:

```
void task::destroy(task& victim)
```

6.7.4. Планирование выполнения логических задач

В каждый момент времени задача может находиться в одном из 5 состояний. Состояние задачи изменяется при вызове методов библиотеки или в результате выполнения определенных действий (например, завершение выполнения метода **task::execute**). В библиотеке TBB реализован метод **task::state**, который возвращает текущее состояние задачи, для которой он был вызван. Данная информация может оказаться полезной при отладке приложений. Прототип метода **task::state** представлен ниже:

```
state_type task::state()
```

Перечислимый тип **task::state_type** может принимать одно из следующих значений, которые отражают текущее состояние выполнения задачи:

- **allocated** – задача только что была создана или был вызван один из методов **task::recycle_***;
- **ready** – задача находится в пуле готовых к выполнению задач или в процессе перемещения в/из него;
- **executing** – задача выполняется и будет уничтожена после завершения метода **task::execute**;
- **freed** – задача находится во внутреннем списке библиотеки свободных задач или в процессе перемещения в/из него;
- **reexecute** – задача выполняется и будет повторно запущена после завершения метода **task::execute**.

Для удобства работы с набором задач библиотекой поддерживается класс **tbb::task_list**. Этот класс фактически представляет собой контейнер задач. Класс **tbb::task_list** содержит два основных метода:

- **task_list::push_back(task& task)** – добавляет задачу в конец списка;

- **task& task_list::pop_front()** – извлекает задачу из начала списка.

Ниже представлены основные методы управления планированием и синхронизации задач:

- **void task::set_ref_count(int count)** – устанавливает значение поля **refcount** равным **count**;
- **void task::wait_for_all()** – ожидает завершения всех подчиненных задач. Поле **refcount** должно быть равно числу подчиненных задач + 1;
- **void task::spawn(task& child)** – добавляет задачу в очередь готовых к выполнению и возвращает управление программному коду, который вызвал этот метод. Задачи **this** и **child** должны принадлежать потоку, который вызывает метод **spawn**. Поле **child.refcount** должно быть больше нуля. Перед вызовом метода **spawn** необходимо с помощью метода **task::set_ref_count** установить число подчиненных задач у задачи **parent.**

  Пример использования рассмотренных методов представлен ниже:

```
task* MyTask::execute()
{
  // ...
  MyTask &t = *new (allocate_child()) MyTask();
  set_ref_count(ref_count() + 2);
  spawn(t);
  wait_for_all();
  // ...
}
```

Продолжаем перечисление методов:

- **void task::spawn (task_list& list)** – добавляет список задач **list** в пул готовых к выполнению и возвращает управление программному коду, который вызвал этот метод. Алгоритм работы данного метода совпадает с последовательным вызовом метода **spawn** для каждой задачи из списка, но имеет более эффективную реализацию. Все задачи из списка **list** и задача **this** должны принадлежать потоку, который вызывает метод **task::spawn**. Поле **child.refcount** должно быть больше нуля для всех задач из списка. Значение поля **depth** у всех задач из списка должно быть одинаковым;
- **void task::spawn_and_wait_for_all(task& child)** – добавляет задачу в очередь готовых к выполнению и ожидает завершения всех подчиненных задач. Является аналогом последовательного вызова методов **spawn** и **wait_for_all**, но имеет более эффективную реализацию;
- **void task::spawn_and_wait_for_all(task_list& list)** – добавляет список задач **list** в очередь готовых к выполнению и ожидает завершения всех подчиненных задач. Является аналогом последовательного вызова методов **spawn** и **wait_for_all**, но имеет более эффективную реализацию;
- **static void task::spawn_root_and_wait(task& root)** – выполняет запуск задачи **root.** Память для задачи должна быть выделена с помощью **task::allocate_root()**. Пример использования этого метода представлен ниже:

```
int main
{
  //...
  MyTask &t = *new (task::allocate_root()) MyTask();
  task::spawn_root_and_wait(t);
  //...
}
```

- **static void task::spawn_root_and_wait(task_list& root_list)** – выполняет параллельный (если возможно) запуск каждой задачи из списка **root_list**, с помощью метода **spawn_root_and_wait.**

Библиотека предоставляет набор методов, которые позволяют повторно использовать задачи для вычислений, что способствует многократному использованию, выделенных ресурсов и уменьшению накладных расходов:

- **void task::recycle_as_continuation()** – изменяет состояние задачи на **allocated**, таким образом после завершения метода **task::execute** задача не уничтожается, а остается в пуле готовых к выполнению. Метод должен быть вызван в теле метода **task::execute**. Значение поля **refcount** должно быть равно числу подчиненных задач и после того, как метод **task::execute** закончит выполнение, должно быть больше нуля (все потомки не должны закончить выполнение). Если это обеспечить нельзя, то необходимо использовать метод **task::recycle_as_safe_continuation**;
- **void task::recycle_as_safe_continuation()** – аналогичен по функциональности методу **task::recycle_as_continuation.** Значение поля **refcount** должно быть равно числу подчиненных задач + 1. Метод должен быть вызван в теле метода **task::execute**;
- **void task::recycle_as_child_of(task& parent)** – устанавливает текущую задачу подчиненной для **parent.** После завершения метода **task::execute** задача не уничтожается, а остается в пуле готовых к выполнению. Этот метод должен быть вызван в теле метода **task::execute**. Пример использования этого метода представлен ниже:

```
task* MyTask::execute()
{
  //...
  empty_task& t = *new( allocate_continuation()) empty_task;
  recycle_as_child_of(t);
  t.set_ref_count(1);
  //...
}
```

- **void task::recycle_to_reexecute()** – запускает текущую задачу на повторное выполнение после завершения выполнения метода **task::execute**. Метод должен быть вызван в теле метода **task::execute**. Метод **task::execute** должен вернуть указатель на другую (не равную **this**) задачу.

В тех случаях когда необходимо изменить алгоритм планирования, может оказаться полезным изменение значения поля **depth**. Для этого используются следующие методы:

- **depth_type task::depth()** – возвращает текущее значение поля **depth**;

- **`void task::set_depth(depth_type new_depth)`** – устанавливает значение поля **`depth`** равным **`new_depth`**. Значение **`new_depth`** должно быть неотрицательным;
- **`void task::add_to_depth(int delta)`** – устанавливает значение поля **`depth`** равным **`depth+delta`**. Значение **`depth+delta`** должно быть неотрицательным.

Ниже представлены методы класса **`tbb::task`**, которые обеспечивают связь задач и потоков, на которых они выполняются:

- **`static task& task::self()`** – возвращает задачу, принадлежащую текущему потоку;
- **`task* task::parent()`** – возвращает значение поля **`parent`**. Для задач, созданных с помощью **`task::allocate_root()`**, значение поля **`parent`** не определено;
- **`bool task::is_stolen_task()`** – возвращает **`true`**, если у задач **`this`** и **`parent`** значения полей **`owner`** не совпадают.

6.7.5. Распараллеливание рекурсии

Одним из достоинств задач является то, что с их помощью можно достаточно легко реализовывать параллельные версии рекурсивных вычислений. Продемонстрируем это на примере «задачи о ферзях». Постановка задачи состоит в следующем. Пусть дана шахматная доска размером **`n`** на **`n`**. Каждый ферзь «бьет» все фигуры, расположенные по горизонтали, вертикали и обеим диагоналям. Необходимо подсчитать число возможных вариантов размещения **`n`** ферзей на этой доске так, чтобы они не «били» друг друга. Программный код, решающий эту задачу обычным перебором с возвратом, представлен ниже:

```
class Backtracker: public task
{
private:
  concurrent_vector<int> placement; // Размещение ферзей. Ферзь
placement[i]
                                    // расположен на i-й вертикали
  int position;                     // Позиция ферзя для проверки
  const int size;                   // Размер поля
  static spin_mutex myMutex;        // Мьютекс для блокировки доступа к
                                    // переменной count
public:
  static int count; // Число вариантов размещения ферзей на доске

public:
  Backtracker(concurrent_vector<int> &t_placement,
    int t_position, int t_size):
    placement(t_placement), position(t_position), size(t_size)
  {}

  task* execute()
  {
    for (int i = 0; i < placement.size(); i++)
      // Проверка горизонтальных и диагональных траекторий на
      // пересечение нового ферзя с уже стоящими
      if ((placement[i] == position) ||
```

```
          (position - placement.size()) == (placement[i] - i) ||
          (position + placement.size()) == (placement[i] + i))
        return NULL;

    placement.push_back(position); // Позицию можно добавить
    if(placement.size() == size)   // Расстановка ферзей на всем
                                   // поле получена
    {
      spin_mutex::scoped_lock lock(myMutex);
      count++;
    }
    else
    {
      empty_task& c = *new(allocate_continuation()) empty_task;
      recycle_as_child_of(c);
      c.set_ref_count(size);

      for (int i = 0; i < size - 1; i++)
      {
        Backtracker& bt =
          *new (c.allocate_child()) Backtracker(placement, i,
                size);
        c.spawn(bt);
      }
      position = size - 1; // Используем текущую "задачу" в
                           // очередной итерации
      return this;
    }

    return NULL;
  }
};

spin_mutex Backtracker::myMutex;
int Backtracker::count;

void Solve(int size)
{
  concurrent_vector<int> placement;
  task_list tasks;

  Backtracker::count=0;
  for (int i = 0; i < size; i++)
  {
    Backtracker& bt =
      *new (task::allocate_root()) Backtracker(placement, i,
            size);
    tasks.push_back(bt);
  }
  task::spawn_root_and_wait(tasks);
}
```

Использованные в коде для синхронизации мьютексы описаны в следующем разделе.

6.8. Примитивы синхронизации

Одной из основных задач при написании параллельных программ является задача синхронизации. При работе приложения в несколько потоков могут возникать ситуации, при которых один поток «ожидает» данных (результатов вычислений) от другого. В этом случае появляется потребность в синхронизации выполнения потоков.

Рассмотрим типичную ситуацию, в которой необходима синхронизация, называемую гонкой данных. Пусть есть общая переменная **`data`**, доступная нескольким потокам для чтения и записи. Каждый поток должен выполнить инкремент этой переменной (то есть выполнить код **`data++`**). Для этого процессору необходимо выполнить три операции: чтение значения переменной из оперативной памяти в регистр процессора, инкремент регистра, запись посчитанного значения в переменную (оперативную память).

Возможны две реализации (с точностью до перестановки потоков) одновременного выполнения такого кода двумя потоками. Наиболее ожидаемое поведение приложения представлено на рис. 9. Сначала поток 0 выполняет чтение переменной, инкремент регистра и запись его значения в переменную, а потом поток 1 выполнит ту же последовательность действий. Таким образом, после завершения работы приложения значение общей переменной будет равно **`data+2`**.

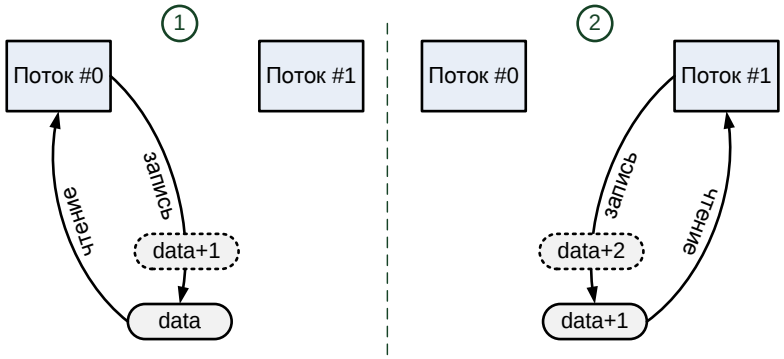

Рис. 9. Вариант реализации гонки данных

Другое возможное поведение представлено на рис. 10. Поток 0 выполняет чтение значения переменной в регистр и инкремент этого регистра, и в этот же момент времени поток 1 выполняет чтение переменной **`data`**. Так как для каждого потока имеется свой набор регистров, то поток 0 продолжит выполнение и запишет в переменную значение **`data+1`**. Поток 1 также выполнит инкремент регистра (значение переменной **`data`** было прочитано из оперативной памяти до записи потоком 0 значения **`data+1`**) и сохранит значение **`data+1`** (свое) в общую переменную. Таким образом, после завершения работы приложения значение переменной будет равно **`data+1`**. Обе реализации могут наблюдаться как на многоядерной (многопроцессорной) системе, так и на однопроцессорной, одноядерной.

Таким образом, в зависимости от порядка выполнения команд результат работы приложения может меняться. Возникает необходимость в механизме синхронизации выполнения потоков, с помощью которого можно было бы обеспечить выполнение части кода не более чем одним потоком в каждый момент времени.

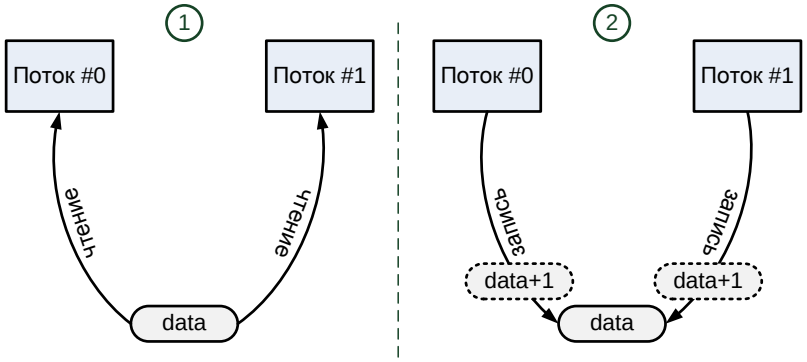

Рис. 10. Вариант реализации гонки данных

Основным способом решения задачи синхронизации является использование примитивов синхронизации. Одним из таких примитивов является *мьютекс* (*mutex*), который реализован в библиотеке TBB.

Мьютекс предназначен для того, чтобы критичный (требующий синхронизации) участок программного кода выполнялся ровно одним потоком. Мьютекс может находиться в одном из двух состояний: «свободен» и «захвачен». Любой поток может захватить мьютекс, переведя его из состояния «свободен» в состояние «захвачен». Если какой-либо поток пытается захватить мьютекс, находящийся в состоянии «захвачен», то выполнение программного кода приостанавливается до тех пор, пока мьютекс не будет переведен в состояние «свободен».

Библиотека TBB содержит реализацию трех типов мьютексов:

- **`mutex`** – мьютекс операционной системы. Представляет собой обертку над примитивами синхронизации операционной системы (в операционных системах семейства Microsoft Windows в качестве основы используются критические секции). Так как данный тип мьютекса реализуется с помощью объектов операционной системы, то поток, пытающийся захватить мьютекс, который находится в состоянии «захвачен», переходит в состояние ожидания, а операционная система передает управление другим потокам. После освобождения мьютекса операционная система переводит поток в состояние готового к выполнению. Таким образом, время, прошедшее после освобождения мьютекса до того момента, когда поток получит управление, может оказаться достаточно большим;
- **`spin_mutex`** – мьютекс, выполняющий активное ожидание. Поток, который пытается захватить этот тип мьютекса, остается активным. Если поток пытается захватить мьютекс, который находится в состоянии «захвачен», то он делает это до тех пор, пока мьютекс не освободится. Таким образом, сразу после освобождения мьютекса один из ожидающих потоков начнет выполнение критического кода. Этот тип мьютекса желательно использовать, когда время выполнения критического участка кода мало;
- **`queuing_mutex`** – этот тип мьютекса выполняет активное ожидание захвата мьютекса с сохранением очередности потоков, т.е. выполнение потоков продолжается в том порядке, в котором они пытались захватить мьютекс. Этот тип мьютексов обычно является наиболее медленным, т.к. его работа несет дополнительные накладные расходы.

Каждый тип мьютексов реализован в виде класса. Обозначим через **M** класс, реализующий мьютекс. Класс **M** содержит всего два метода:

- **M::M()** – конструктор. Создает мьютекс, находящийся в «свободном» состоянии;
- **M::~M()** – деструктор. Уничтожает мьютекс, находящийся в «свободном» состоянии.

Для захвата/освобождения мьютекса предназначен класс **scoped_lock**, реализованный в теле каждого класса мьютекса. Класс **scoped_lock** содержит следующие методы:

- **M::scoped_lock::scoped_lock()** – конструктор. Создает объект класса **scoped_lock** без захвата мьютекса.
- **M::scoped_lock::scoped_lock(M&)** – конструктор с параметром. Создает объект класса **scoped_lock** и переводит мьютекс в состояние «захвачен».
- **M::scoped_lock::~scoped_lock()** – деструктор. Переводит мьютекс в состояние «свободен», если он был захвачен.
- **M::scoped_lock::acquire(M&)** – метод захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен». Если мьютекс уже находится в состоянии «захвачен», то поток блокируется.
- **bool M::scoped_lock::try_acquire(M&)** – метод неблокирующего захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен», метод возвращает **true**. Если мьютекс уже находится в состоянии «захвачен», то ничего не происходит, метод возвращает **false**.
- **M::scoped_lock::release()** – метод освобождения мьютекса. Переводит мьютекс в состояние «свободен», если он был захвачен.

Пример, выполняющий подсчет количества вызовов метода **operator()** с использованием мьютексов, представлен ниже:

```
int Functor::count = 0;
mutex Functor::myMutex;

class Functor
{
private:
  static int count;
  static mutex SumMutex;

public:
  // Методы функтора
  //...
  void operator()(const blocked_range<int>& Range) const
  {
    mutex::scoped_lock lock;
    lock.acquire(myMutex);
    count++;
    lock.release();
  }
};
```

Помимо обычных мьютексов в библиотеке TBB реализованы мьютексы читателей-писателей. Эти мьютексы содержат дополнительный флаг **writer**. С помо-

щью этого флага можно указать, какой тип блокировки мьютекса нужно захватить: читателя (**writer=false**) или писателя (**writer=true**). Несколько «читателей» (потоки, которые захватили блокировку читателя) при отсутствии блокировки писателя могут выполнять критичный код одновременно. Если поток захватил мьютекс писателя, то все остальные потоки блокируются при попытке захвата мьютекса.

Библиотека содержит два типа мьютексов читателей-писателей: **spin_rw_mutex** и **queuing_rw_mutex**. Эти мьютексы по своим характеристикам полностью совпадают с мьютексами **spin_mutex** и **queuing_mutex**. Отличия мьютексов читателей-писателей от остальных заключаются только в методах класса **scoped_lock.**

Класс **scoped_lock** для мьютексов читателей-писателей содержит следующие методы:

- **M::scoped_lock::scoped_lock()** – конструктор. Создает объект класса **scoped_lock** без захвата мьютекса.
- **M::scoped_lock::scoped_lock(M&, bool write=true)** – конструктор с параметром. Создает объект класса **scoped_lock** и переводит мьютекс в состояние «захвачен». Второй параметр указывает тип блокировки, которую необходимо захватить: читателя (**writer=false**) или писателя (**writer=true**).
- **M::scoped_lock::~scoped_lock()** – деструктор. Переводит мьютекс в состояние «свободен», если он был захвачен.
- **M::scoped_lock::acquire(M&, bool write=true)** – метод захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен». Второй параметр указывает тип блокировки, которую необходимо захватить: читателя (**writer=false**) или писателя (**writer=true**). Если мьютекс уже находится в состоянии «захвачен», то поток блокируется.
- **bool M::scoped_lock::try_acquire(M&, bool write=true)** – метод неблокирующего захвата мьютекса. Второй параметр указывает тип блокировки, которую необходимо захватить: читателя (**writer=false**) или писателя (**writer=true**). Переводит мьютекс из состояния «свободен» в состояние «захвачен», метод возвращает **true**. Если мьютекс уже находится в состоянии «захвачен», то ничего не происходит, метод возвращает **false.**
- **M::scoped_lock::release()** – метод освобождения мьютекса. Переводит мьютекс в состояние «свободен», если он был захвачен.
- **M::scoped_lock::upgrade_to_writer()** – изменяет тип блокировки на блокировку писателя.
- **M::scoped_lock::downgrade_to_reader()** – изменяет тип блокировки на блокировку читателя.

Многие функции и классы библиотеки TBB выполняют синхронизацию неявно (без использования мьютексов). Например, функция **parallel_for** «ждет» завершения всех потоков, занятых вычислениями, прежде чем вернуть управление потоку, с которого она была вызвана.

Помимо мьютексов библиотека TBB содержит шаблонный класс **tbb::atomic**, который также может использоваться для синхронизации. Методы этого класса являются атомарными, т.е. несколько потоков не могут одновременно выполнять методы этого класса. Если потоки пытаются одновременно выполнить ме-

тоды класса **tbb::atomic**, то один из потоков блокируется и ожидает завершения выполнения метода другим. Пример, выполняющий подсчет количества вызовов метода **operator()** с использованием класса **tbb::atomic**, представлен ниже:

```
atomic<int> Functor::count;

class Functor
{
private:
  static atomic<int> count;

public:
  // Методы функтора
  //...
  void operator()(const blocked_range<int>& Range) const
  {
    count++;
  }
};
```

6.9. Потокобезопасные контейнеры

Библиотека TBB содержит реализацию трех контейнеров, которые являются аналогами контейнеров STL:

- **tbb::concurrent_queue** – очередь;
- **tbb::concurrent_hash_map** – хеш-таблица;
- **tbb::concurrent_vector** – вектор.

В качестве примера рассмотрим **tbb::concurrent_queue**. Так как контейнеры библиотеки TBB предназначены для параллельной работы, то их интерфейс отличается от интерфейса STL-контейнеров. Отличия между **std::queue** и **tbb::concurrent_queue** заключаются в следующем:

- Методы **queue::front** и **queue::back** не реализованы в **tbb::concurrent_queue**, т.к. их параллельное использование может быть небезопасно.
- В отличие от STL тип **size_type** является знаковым.
- Метод **concurrent_queue::size** возвращает разницу между числом операций добавления элементов и операций извлечения элементов. Если в момент вызова этого метода выполняются методы добавления/извлечения методов, то они тоже учитываются.
- Для **tbb::concurrent_queue** разрешен вызов метода **pop** для пустой очереди. После его вызова поток блокируется до тех пор, пока в очередь не положат элемент.
- **tbb::concurrent_queue** содержит метод **pop_if_present**, который извлекает элемент из очереди, если в очереди есть элементы.

Более подробная информация о потокобезопасных контейнерах библиотеки TBB представлена в [3].

6.10. Литература

Использованные источники

1. Официальный сайт Intel® Threading Building Blocks. – [http://www.intel.com/software/products/tbb/]
2. Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
3. Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.

Рекомендуемая литература

4. Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley, 2000 (русский перевод: Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).
5. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004.
6. Chapman B., Jost G., van der Pas R. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific Computation and Engineering). 2007.
7. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. – СПб.: Питер, 2006.
8. Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2006.
9. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб.: Питер, 2003.
10. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows: Пер. с англ. – 4-е изд. – СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.

Информационные ресурсы сети Интернет

11. Страница библиотеки TBB на сайте корпорации Intel – [http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm].
12. Сайт сообщества пользователей TBB – [http://threadingbuildingblocks.org].
13. Сайт Лаборатории параллельных информационных технологий НИВЦ МГУ –[http://www.parallel.ru].
14. Официальный сайт OpenMP – [www.openmp.org].

6.11. Приложения

6.11.1. Заголовочные файлы библиотеки TBB

Для того чтобы воспользоваться классами и функциями библиотеки TBB, необходимо подключать соответствующие заголовочные файлы. Например, для работы с функцией **`parallel_for`** необходимо подключить файл **`parallel_for.h`**, для работы с **`concurrent_hash_map`** – **`concurrent_hash_map.h`**. Полная таблица функций/классов и необходимых для них заголовочных файлов представлена ниже.

Название функции/класса	**Заголовочный файл**
aligned_space	aligned_space.h
atomic	atomic.h
blocked_range	blocked_range.h
blocked_range2d	blocked_range2d.h
cache_aligned_allocator	cache_aligned_allocator.h
concurrent_hash_map	concurrent_hash_map.h
concurrent_queue	concurrent_queue.h
concurrent_vector	concurrent_vector.h
tick_count	tick_count.h
mutex	mutex.h
parallel_for	parallel_for.h
parallel_reduce	parallel_reduce.h
parallel_scan	parallel_scan.h
parallel_sort	parallel_sort.h
parallel_while	parallel_while.h
partitioner	partitioner.h
pipeline	pipeline.h
queuing_mutex	queuing_mutex.h
queuing_rw_mutex	queuing_rw_mutex.h
scalable_allocator	scalable_allocator.h
spin_mutex	spin_mutex.h
spin_rw_mutex	spin_rw_mutex.h
split	tbb_stddef.h
task	task.h
task_scheduler_init	task_scheduler_init.h

6.11.2. Сборка и настройка проекта

Для сборки приложения, использующего библиотеку TBB**,** необходимо указать библиотеку, с которой будет линковаться приложение: **`tbb_debug.lib`** или

tbb.lib. Библиотека **tbb_debug.lib** выполняет проверки корректности во время выполнения приложения и полностью поддерживается профилировщиком Intel® Thread Profiler. Библиотека **tbb.lib** имеет гораздо более эффективную реализацию функций и методов, чем **tbb_debug.lib**. При отладке приложения рекомендуется использовать библиотеку **tbb_debug.lib**, при сборке рабочей версии – **tbb.lib**. При использовании в приложении операторов, аллокаторов выделения динамической памяти необходимо указать библиотеку **tbbmalloc.lib** (или **tbbmalloc_debug.lib**). Для подключения библиотеки в **Microsoft Visual Studio 2005** необходимо выполнить следующую последовательность действий:

1. В меню **Tools** выберите пункт **Options…**. В открывшемся окне выберите **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выберите пункт **Library files**. Нажмите левой кнопкой мыши на изображении папки и укажите путь к папке **lib** библиотеки **Intel Threading Building Blocks**. При установке по умолчанию этот путь будет **C:\Program Files\Intel\TBB\2.0\<arch>\vc8\lib** (где **<arch>** должно быть **ia32** или **em64t** в зависимости от режима работы процессора). Нажмите **OK**.
2. В окне **Solution Explorer** нажмите правой кнопкой мыши на названии проекта и выберите пункт **Properties**.
3. Выберите пункт **Linker\Input** и в поле **Additional Dependencies** введите название библиотеки: **tbb_debug.lib** для **Debug**-сборки или **tbb.lib** для **Release**-сборки.

Для работы приложения, использующего библиотеку TBB, необходимо иметь одну динамическую библиотеку: **tbb.dll** (при использовании **tbb.lib**) или **tbb_debug.dll** (при использовании **tbb_debug.lib**). При использовании в приложении операторов выделения динамической памяти (аллокаторов) необходимо иметь динамическую библиотеку **tbbmalloc.dll** (при использовании **tbbmalloc.lib**) или **tbbmalloc_debug.dll** (при использовании **tbbmalloc_debug.lib**).

6.11.3. Совместное использование с OpenMP

Библиотеку TBB можно использовать одновременно с OpenMP. Для этого на каждом потоке, созданном с помощью OpenMP (внутри параллельной секции), необходимо запустить планировщик потоков TBB.

```
int main()
{
  #pragma omp parallel
  {
    task_scheduler_init init;
    #pragma omp for
    for( int i=0; i<n; i++ )
    {
      // Можно использовать функции и классы библиотеки TBB
    }
  }
}
```

6.11.4. Оценка эффективности приложений

Основным способом оценки эффективности приложения является измерение времени выполнения приложением вычислительно трудоемких операций. Библиотека TBB содержит класс **tbb::tick_count**, с помощью которого можно выполнять измерения времени. Этот класс реализован таким образом, что независимо от аппаратной конфигурации (многопроцессорности, многоядерности) измеряемое время является синхронным между потоками. В операционных системах семейства Microsoft Windows класс **tbb::tick_count** реализован с использованием функции **QueryPerformanceCounter.**

Основной метод класса **tbb::tick_count** – метод **now**, измеряющий текущее значение времени. Этот метод является статическим и возвращает экземпляр класса **tbb::tick_count**. Его прототип представлен ниже:

```
static tick_count tick_count::now()
```

Выполнив два замера времени (в начале и в конце измеряемого участка программы), необходимо вычесть два экземпляра класса **tbb::tick_count** и перевести интервал времени в секунды. Результатом вычитания двух экземпляров класса **tbb::tick_count** является экземпляр вспомогательного класса **tick_count::interval_t**, который представляет интервал времени. Для перевода интервала времени в секунды класс **tick_count::interval_t** содержит метод **seconds**. Его прототип представлен ниже:

```
double interval_t::seconds()
```

Типичная схема измерения времени выполнения вычислений представлена ниже:

```
void SomeFunction()
{
  tick_count t0 = tick_count::now();
  // Вычисления
  tick_count t1 = tick_count::now();
  printf("Время вычислений = %f seconds\n", (t1 - t0).seconds());
}
```

6.11.5. Динамическое выделение памяти

Обычные операторы выделения динамической памяти работают с общей кучей для всех потоков, что требует наличия синхронизации. Библиотека TBB содержит масштабируемые операторы выделения динамической памяти (аллокаторы) на каждый поток. Прототипы функций выделения динамической памяти для С-программ представлены ниже:

```
void* scalable_calloc(size_t nobj, size_t size);
void  scalable_free(void* ptr);
void* scalable_malloc(size_t size);
void* scalable_realloc(void* ptr, size_t size);
```

Более подробная информация о динамическом выделении памяти представлена в [2].

7. Лабораторная работа. Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения

7.1. Введение

В настоящей лабораторной работе рассматривается один из инструментов параллельного программирования, предназначенный для распараллеливания решения задач в системах с общей памятью, – библиотека Intel Threading Building Blocks (TBB) [11]. Основная идея, заложенная в библиотеку, состоит в использовании стандартного высокоуровневого C++ для быстрой разработки кросс-платформенных, хорошо масштабируемых параллельных приложений (см. подробнее в [1]). Наряду с указанным выше, использование библиотеки TBB предоставляет механизмы абстрагирования от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении прикладной задачи, что является достаточно актуальным.

Изучение принципов функционирования и вопросов эффективного использования TBB проводится в данной лабораторной работе на примере задачи матрично-векторного умножения, ставшей одной из классических задач для формирования навыков параллельного программирования. При этом основной упор делается на ознакомление с распараллеливанием циклов.

В ходе выполнения работы предполагается, что слушатели имеют навыки разработки объектно-ориентированных программ на C++, а также владеют основами параллельного программирования в системах с общей памятью.

7.2. Методические рекомендации

7.2.1. Цели и задачи работы

Целью данной лабораторной работы является приобретение практических навыков распараллеливания циклов для систем с общей памятью при помощи библиотеки TBB.

Данная цель предполагает решение следующих задач:

- изучение способов инициализации и завершения работы библиотеки TBB (см. также [1]);
- освоение функциональности библиотеки TBB, связанной с распараллеливанием циклов (см. также [1]);
- рассмотрение учебной задачи, направленной на демонстрацию принципов распараллеливания циклов при помощи библиотеки TBB;
- самостоятельная разработка и отладка параллельных программ, реализующих распараллеливание циклов при помощи TBB.

7.2.2. Структура работы

Лабораторная работа состоит из введения, двух разделов, списка дополнительных заданий и списка литературы. Во введении обосновывается актуальность использования TBB в процессе разработки параллельных программ. В первом раз-

деле приводятся методические рекомендации к лабораторной работе: формулируются цели и задачи, системные требования, рекомендации по проведению занятий. Во втором разделе изучается применение библиотеки ТВВ для распараллеливания циклов. Изучение проводится на примере задачи матрично-векторного умножения. В заключение приводятся задания для самостоятельной проработки, а также использованная и рекомендуемая литература.

Рассмотрение задачи матрично-векторного умножения проводится в соответствии с материалами учебного курса [8], разработанного в ННГУ (руководитель авторского коллектива д.т.н., проф. В.П. Гергель).

7.2.3. Системные требования

Приведем системные требования для Windows-систем. Аналогичную информацию для систем на базе Linux и Mac OS можно найти на официальном сайте ТВВ [11].

Аппаратное обеспечение

Минимальные требования

- Intel® Pentium® 4 процессор;
- 512 Мб ОЗУ;
- 300 Мб дискового пространства.

Рекомендуемые требования

- Intel Pentium 4 процессор с поддержкой технологии Hyper-Threading (HT Technology) или Intel® Xeon® процессор;
- 1 Гб ОЗУ.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

Программное обеспечение

- Microsoft Windows XP Professional, Microsoft Windows Server 2003 или Microsoft Windows Vista;
- Intel® C++ Compiler 9.0 for Windows или старше;
- Microsoft Visual C++ 7.1 или старше;
- Microsoft Internet Explorer 6.0 или старше;
- Adobe Reader 6.0 или старше.

7.2.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется придерживаться следующей последовательности изучения материала:

1. Изучить способы инициализации и завершения работы библиотеки ТВВ (см. также [1]).
2. Рассмотреть функциональность библиотеки ТВВ, связанную с распараллеливанием циклов (см. также [1]).

3. Разобрать один из вариантов решения учебной задачи – задачи умножения матрицы на вектор, демонстрирующей принципы распараллеливания циклов при помощи библиотеки TBB.
4. Перейти к выполнению дополнительных заданий, состоящих в самостоятельной разработке и отладке параллельных программ, реализующих распараллеливание циклов при помощи TBB.

7.3. Постановка задачи[8]

Имея несложную математическую постановку, задача матрично-векторного умножения является классической задачей для приобретения навыков разработки, отладки и оптимизации как последовательных, так и параллельных программ. Такая популярность задачи вызвана, прежде всего, богатыми возможностями демонстрации типовых подходов к организации процесса вычислений, а также эффективной реализации этих подходов средствами языка программирования. В данной лабораторной работе мы рассмотрим один из возможных подходов к решению задачи, его реализацию в виде последовательной программы на C++, а также на его примере познакомимся с принципами распараллеливания при помощи библиотеки TBB. При этом в основном будет затронута часть функциональности TBB, связанная с распараллеливанием циклов.

В результате умножения матрицы A размерности $m \times n$ и вектора b, состоящего из n элементов, получается вектор c размера m, каждый i-й элемент которого есть результат скалярного умножения i-й строки матрицы A (обозначим эту строчку a_i) и вектора b:

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j \ , 1 \le i \le m.$$

Тем самым, получение результирующего вектора c предполагает повторение m однотипных операций по умножению строк матрицы A и вектора b. Каждая такая операция включает умножение элементов строки матрицы и вектора b (n операций) и последующее суммирование полученных произведений ($n - 1$ операций). Общее количество необходимых скалярных операций есть величина $T_1 = m(2n - 1)$ [8].

7.4. Последовательный алгоритм

Спецификации последовательного алгоритма умножения матрицы на вектор могут быть представлены следующим образом:

Исходные данные: A[m][n] – матрица размерности $m \times n$.
b[n] – вектор, состоящий из n элементов.

[8] Часть данного материала, посвященная созданию последовательного приложения (задания 1–5 раздела 7.5), написана на основе раздела 7 курса [1], а также лабораторной работы «Умножение матрицы на вектор» с незначительными изменениями, преимущественно касающимися другого способа реализации замеров времени. Алгоритм параллельного умножения матрицы на вектор, рассматривающий матрицу как набор строк, изложен на основе раздела 7 курса [1]. Наличие подобных материалов в данной лабораторной работе прежде всего обосновано возможностью ее использования как отдельного методического материала.

Результат: c[n] – вектор из *m* элементов.

Алгоритм вычисления результата матрично-векторного умножения полностью описывается его формулой, поэтому сразу приведем код:

```
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++)
{
  c[i] = 0;
  for (j = 0; j < n; j++)
  {
    c[i] += A[i][j] * b[j];
  }
}
```

Матрично-векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины *n* требует выполнения *n* операций умножения и *n*–1 операций сложения, его трудоемкость порядка *O*(*n*). Для выполнения матрично-векторного умножения необходимо выполнить *m* операций вычисления скалярного произведения, таким образом, алгоритм имеет трудоемкость порядка *O*(*mn*).

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем предполагать, что матрица *A* является квадратной, т.е. *m* = *n*.

Проведем пошагово реализацию последовательной программы умножения матрицы на вектор. Заметим, что вариант, который будет рассмотрен, разумеется, не является единственным. Будем также считать основной целью корректность работы полученной реализации, а не ее производительность на какой-либо конкретной архитектуре.

7.5. Реализация последовательного алгоритма

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матрично-векторного умножения. Начальный вариант будущей программы представлен в проекте **`SerialMatrixVector`**, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе данного упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера объектов, инициализации матрицы и вектора, умножения матрицы на вектор и вывода результатов.

Задание 1. Открытие проекта SerialMatrixVector

Откройте проект **SerialMatrixVector**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLabs\Serial MatrixVector**;
- дважды щелкните на файле **SerialMatrixVector.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialMV.cpp**, как это показано на рис. 1. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

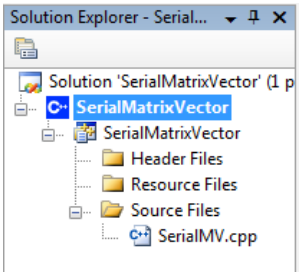

Рис. 1. Открытие файла **SerialMV.cpp**

В файле **SerialMV.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции **`main`**. Эта заготовка содержит объявление переменных и вывод на печать начального сообщения программы.

Рассмотрим переменные, которые используются в основной функции (**`main`**) нашего приложения. Первые две из них (**`pMatrix`** и **`pVector`**) – это, соответственно, матрица и вектор, которые участвуют в матрично-векторном умножении в качестве аргументов. Третья переменная **`pResult`** – вектор, который должен быть получен в результате матрично-векторного умножения. Переменная **`Size`** определяет размер матриц и векторов (предполагаем, что матрица **`pMatrix`** квадратная, имеет размерность **`Size`**×**`Size`** и умножается на вектор из **`Size`** элементов). Далее объявлены переменные циклов.

```
double* pMatrix; // Initial matrix
double* pVector; // Initial vector
double* pResult; // Result vector for matrix-vector
                 // multiplication
int Size;        // Sizes of initial matrix and vector
```

Заметим, что для хранения матрицы **`pMatrix`** используется одномерный массив, в котором матрица хранится построчно в соответствии с соглашениями языка C. Таким образом, элемент, расположенный на пересечении *i*-й строки и *j*-го столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.

Программный код, который следует за объявлением переменных, – это вывод начального сообщения и ожидание нажатия любой клавиши перед завершением выполнения приложения:

```
printf ("Serial matrix-vector multiplication program\n");
getch();
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна **Microsoft Visual Studio 2005** появилось сообщение "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

Сразу после запуска кода в командной консоли появится сообщение: "Serial matrix-vector multiplication program". Для того чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2. Ввод размеров объектов

Для задания исходных данных последовательного алгоритма умножения матрицы на вектор реализуем функцию **ProcessInitialization**. Эта функция предназначена для определения размеров матрицы и вектора, выделения памяти для всех объектов, участвующих в умножении (исходной матрицы **pMatrix** и вектора **pVector**, и результата умножения **pResult**), а также для задания значений элементов исходной матрицы и вектора. Значит, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size);
```

На первом этапе необходимо определить размеры объектов (задать значение переменной **Size**). В тело функции **ProcessInitialization** добавьте выделенный фрагмент кода:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size)
{
  // Setting the size of initial matrix and vector
  printf("\nEnter size of the initial objects: ");
  scanf("%d", &Size);
  printf("\nChosen objects' size = %d", Size);
}
```

Пользователю предоставляется возможность ввести размер объектов (матрицы и вектора), который затем считывается из стандартного потока ввода **stdin** и сохраняется в целочисленной переменной **Size**. Далее печатается значение переменной **Size** (рис. 2).

После строки, выводящей на экран приветствие, добавьте вызов функции инициализации процесса вычислений **ProcessInitialization** в тело основной функции последовательного приложения:

```
void main()
{
  double* pMatrix;  // Initial matrix
  double* pVector;  // Initial vector
  // Result vector for matrix-vector multiplication
  double* pResult;
  int Size;         // Sizes of initial matrix and vector
  time_t start, finish;
  double duration;

  printf("Serial matrix-vector multiplication program\n");
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что значение переменной **Size** задается корректно.

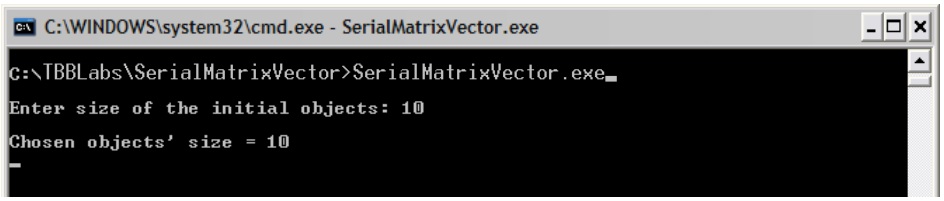

Рис. 2. Задание размера объектов

Теперь обратимся к вопросу контроля правильности ввода. Так, например, если в качестве размера объектов пользователь попытается ввести неположительное число, приложение должно либо завершить выполнение, либо продолжать запрашивать размер объектов до тех пор, пока не будет введено положительное число. Реализуем второй вариант поведения, для этого фрагмент кода, который производит ввод размера объектов, поместим в цикл с постусловием:

```
  // Setting the size of initial matrix and vector
  do
  {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);
```

Снова скомпилируйте и запустите приложение. Попытайтесь ввести неположительное число в качестве размера объектов. Убедитесь в том, что ошибочные ситуации обрабатываются корректно.

Задание 3. Ввод данных

Функция инициализации должна также выделять память для хранения объектов (добавьте выделенный код в тело функции **ProcessInitialization**):

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int Size)
{
  // Setting the size of initial matrix and vector
  do
  {
    <...>
  }
  while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size * Size];
  pVector = new double [Size];
  pResult = new double [Size];
}
```

Далее необходимо задать значения всех элементов матрицы **pMatrix** и вектора **pVector**. Для выполнения этих действий реализуем еще одну функцию **DummyDataInitialization**. Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of matrix and vector
// elements
void DummyDataInitialization(double* pMatrix, double* pVector,
  int Size)
{
  int i, j;  // Loop variables

  for (i = 0; i < Size; i++)
  {
    pVector[i] = 1;
    for (j = 0; j < Size; j++)
      pMatrix[i * Size + j] = i;
  }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матрицы и вектора достаточно простым образом: значение элемента матрицы совпадает с номером строки, в которой он расположен, а все элементы вектора равны 1. То есть в случае когда пользователь выбрал размер объектов, равный 4, будут определены следующие матрица и вектор (формирование исходных данных при помощи датчика случайных чисел будет рассмотрено в задании 6):

$$pMatrix = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}, \; pVector = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Вызов функции **DummyDataInitialization** необходимо выполнить после выделения памяти внутри функции **ProcessInitialization**:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int Size)
{
  // Setting the size of initial matrix and vector
  do
  {
    <...>
  }
  while (Size <= 0);

  // Memory allocation
  <...>

  // Initialization of matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}
```

Реализуем еще две функции, которые помогут контролировать ввод данных, – функции форматированного вывода объектов: **PrintMatrix** и **PrintVector**. В качестве аргументов в функцию форматированной печати матрицы **PrintMatrix** передается указатель на одномерный массив, где эта матрица хранится построчно, а также размеры матрицы по вертикали (количество строк **RowCount**) и горизонтали (количество столбцов **ColCount**). Для форматированной печати вектора при помощи функции **PrintVector** необходимо сообщить функции указатель на вектор, а также количество элементов в нем.

```
// Function for formatted matrix output
void PrintMatrix(double* pMatrix, int RowCount, int ColCount)
{
  int i, j; // Loop variables
  for (i = 0; i < RowCount; i++)
  {
    for (j = 0; j < ColCount; j++)
      printf("%7.4f ", pMatrix[i * ColCount + j]);
    printf("\n");
  }
}

// Function for formatted vector output
void PrintVector(double* pVector, int Size)
{
  int i;
  for (i = 0; i < Size; i++)
    printf("%7.4f ", pVector[i]);
  printf("\n");
}
```

Добавим вызов этих функций в основную функцию приложения:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf("Initial Matrix: \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector: \n");
  PrintVector(pVector, Size);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 3). Выполните несколько запусков приложения, задавая различные размеры объектов.

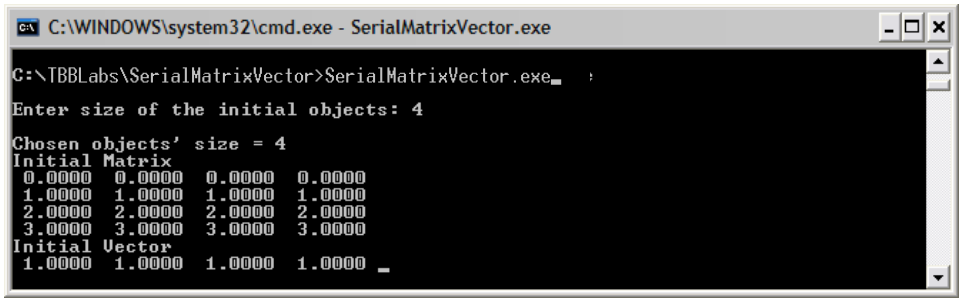

Рис. 3. Результат работы программы при завершении задания 3

Задание 4. Завершение процесса вычислений

Перед выполнением матрично-векторного умножения сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию **ProcessTermination**. Память выделялась для хранения исходных матрицы **pMatrix** и вектора **pVector**, а также для хранения результата умножения **pResult**. Следовательно, эти объекты необходимо передать в функцию **ProcessTermination** в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector,
  double* pResult)
{
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}
```

Вызов функции **ProcessTermination** необходимо выполнить перед завершением той части программы, которая выполняет умножение матрицы на вектор:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);
  printf ("Initial Vector: \n");
  PrintVector (pVector, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

Задание 5. Реализация умножения матрицы на вектор

Выполним теперь разработку основной вычислительной части программы. Для выполнения умножения матрицы на вектор реализуем функцию **SerialResultCalculation**, которая принимает на вход исходные матрицу **pMatrix** и вектор **pVector**, размеры этих объектов **Size**, а также указатель на вектор в памяти, где должен быть сохранен результат **pResult**.

В соответствии с алгоритмом, изложенным ранее, код этой функции должен быть следующий:

```
// Function for matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size)
{
  int i, j;  // Loop variables
  for (i = 0; i < Size; i++)
  {
    pResult[i] = 0;
```

```
    for (j = 0; j < Size; j++)
      pResult[i] += pMatrix[i * Size + j] * pVector[j];
  }
}
```

Следует отметить, что приведенный код может быть оптимизирован, однако такая оптимизация не является целью данного учебного материала и приведет к усложнению программ.

Выполним вызов функции вычисления умножения матрицы на вектор из основной программы. Для контроля правильности выполнения умножения распечатаем результирующий вектор:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf("Initial Matrix: \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector: \n");
  PrintVector(pVector, Size);

  // Matrix-vector multiplication
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

  // Printing the result vector
  printf("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матрицы на вектор. Если алгоритм реализован правильно, то результирующий вектор должен иметь следующую структуру: *i*-й элемент результирующего вектора равен произведению размера вектора **`Size`** на номер элемента *i*. Так, если размер объектов **`Size`** равен 4, результирующий вектор **`pResult`** должен быть таким: **`pResult`** = (0, 4, 8, 12). Проведите несколько вычислительных экспериментов, изменяя размеры объектов.

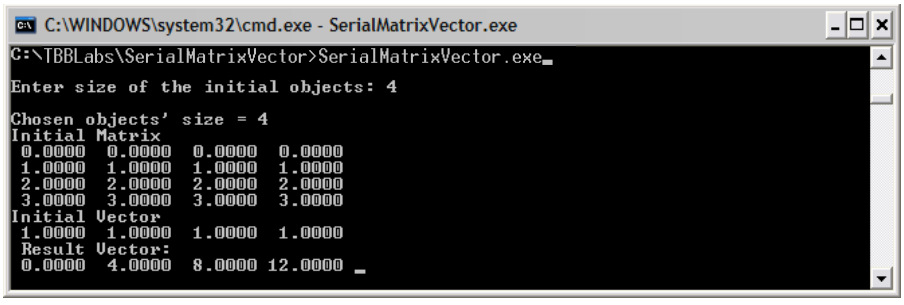

Рис. 4. Результат выполнения матрично-векторного умножения

Задание 6. Проведение вычислительных экспериментов

После реализации параллельной версии алгоритма нам потребуется оценивать ее ускорение. Для этого сейчас необходимо провести вычислительные экспе-

рименты и замерить времена работы последовательной версии. Анализировать время выполнения последовательной реализации разумно для достаточно больших объектов. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов **RandomDataInitialization** (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of objects' elements
void RandomDataInitialization(double* pMatrix,double* pVector,
  int Size)
{
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i = 0; i < Size; i++)
  {
    pVector[i] = rand() / double(1000);
    for (j = 0; j < Size; j++)
    pMatrix[i * Size + j] = rand() / double(1000);
  }
}
```

Заметим, что в отладочных целях следует инициализировать датчик случайных чисел при помощи функции **srand()** одним и тем же числом, что обеспечит нам одинаковые последовательности случайных чисел и позволит обнаружить ошибки, если они есть.

Будем вызывать эту функцию вместо ранее разработанной функции **DummyDataInitialization**, которая генерировала данные так, чтобы можно было легко проверить правильность работы алгоритма:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int Size)
{
  // Size of initial matrix and vector definition
  <...>

  // Memory allocation
  <...>

  // Random data initialization
  RandomDataInitialization(pMatrix, pVector, Size);
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Для определения времени можно воспользоваться стандартной функцией языка С, которая позволяет узнать время работы программы или ее части:

```
time_t clock(void);
```

К сожалению, данный счетчик обладает малой чувствительностью, поэтому он не может быть использован для измерения времени работы программ с малым числом вычислительных операций. Реализуем функцию **GetTime()**, которая позволяет более точно замерить время выполнения участка кода с использованием функций библиотеки WinAPI:

```
// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x)
{
double result =
  ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
  return result;
}

// Function that gets the timestamp in seconds
double GetTime()
{
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency(&lpFrequency);
  QueryPerformanceCounter(&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount) / LiToDouble(lpFrequency);
}
```

Функция **GetTime** возвращает текущее значение времени, которое отсчитывается от фиксированного момента в прошлом, в секундах. Следовательно, вызвав эту функцию два раза – до и после исследуемого фрагмента – можно вычислить время его работы. Например, этот фрагмент вычислит время **duration** работы функции **f()**.

```
  double t1, t2;
  t1 = GetTime();
  f();
  t2 = GetTime();
  double duration = (t2 - t1);
```

Отметим, что представленная реализация функции **GetTime** не претендует на оптимальность и может быть улучшена. Сделать это предоставляем читателям самостоятельно.

Библиотека TBB реализует собственный способ замера времени, сочетающий высокую разрешающую способность с простотой использования. Для этого предусмотрен специальный класс **tick_count**, содержащий статический метод **tick_count::now()**, возвращающий текущее время как объект класса **tick_count**. Как и раньше, для измерения времени необходимо вызвать метод **now()** в начале и в конце фрагмента программы с вычислительной частью, после чего вычесть из второго значения первое (операция вычитания для класса **tick_count** перегружена). Полученный результат можно перевести в секунды при помощи метода **tick_count::seconds()**.

Отметим, что в реализации библиотеки TBB для ОС Windows функция **tick_count::now()** вызывает использованный нами **QueryPerformance Counter**.

Чтобы воспользоваться описанным способом замера времени, необходимо подключить заголовочный файл tbb/tick_count.h.

Добавим в программный код вычисление и вывод времени непосредственного выполнения умножения матрицы на вектор, для этого поставим замеры времени до и после вызова функции **SerialResultCalculation**:

```
  tick_count Start, Finish;

  // Matrix-vector multiplication
  Start = tick_count::now();
  SerialResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = tick_count::now();
```

```
Duration = (Finish - Start).seconds();

// Printing the result vector
printf ("\n Result Vector: \n");
PrintVector(pResult, Size);
// Printing the time spent by matrix-vector multiplication
printf("\n Time of serial execution: %f", Duration);
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами отключите печать матриц и векторов (закомментируйте соответствующие строки кода). Убедитесь, что выбрана конфигурация **Release**. Проведите вычислительные эксперименты.

Задание 7. Оценка эффективности

Запустите программу несколько раз, задавая одни и те же размеры исходных данных. Нетрудно видеть, что время работы меняется от запуска к запуску. Эффект вызван особенностями работы программ под управлением многозадачной операционной системы и погрешностью механизма измерения. В связи с этим существуют несколько методик организации замеров времени. Мы в данной лабораторной работе рекомендуем выполнить приложение несколько раз и выбрать минимальное из полученных времен работы при одних и тех же размерах матрицы и вектора.

Проведите эксперименты, результаты занесите в таблицу 1.

Таблица 1

Время работы последовательного приложения

Номер теста	Параметр Size	Время работы последовательного приложения (сек)
1	250	
2	500	
3	1000	
4	2000	
5	4000	
6	8000	
7	10000	

7.6. Параллельный алгоритм[9]

7.6.1. Принципы распараллеливания

Разработка алгоритмов (а в особенности – методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. В нашем случае будем полагать, что вычислительная схема решения задачи умножения матрицы на вектор уже известна. Действия для определения эффективного способа организации параллельных вычислений могут состоять в следующем:

- Выполнить анализ имеющейся вычислительной схемы и осуществить ее разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга.

[9] По материалам лабораторной работы 1 к разделу 7 курса [8].

- Выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи.
- Выполнить распределение выделенных *подзадач* по вычислительным элементам (процессорам, ядрам).

Из общих соображений понятно, что объем вычислений для каждого используемого потока должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*). Кроме того, также понятно, что распределение подзадач между процессорами (ядрами) должно быть выполнено таким образом, чтобы число информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

7.6.2. Определение подзадач

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов, и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Общая характеристика распределения данных для матричных алгоритмов содержится в разделе 7 учебного курса [8]. Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

Далее в лабораторной работе будет рассматриваться *алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами* (*горизонтальными полосами*) *строк*. При таком способе разделения данных в качестве базовой подзадачи может быть выбрана *операция скалярного умножения одной строки матрицы на вектор*.

7.6.3. Выделение информационных зависимостей

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений при ленточной схеме разделения данных показана на рис. 5.

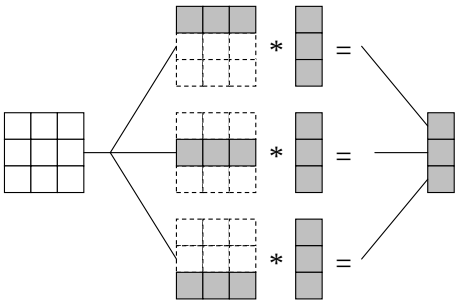

Рис. 5. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

7.6.4. Масштабирование и распределение подзадач по вычислительным элементам

В процессе умножения плотной матрицы, разбитой на строки или столбцы, на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае когда число вычислительных элементов p меньше числа базовых подзадач m ($p < m$), возможно объединение базовых подзадач, для того чтобы каждый вычислительный элемент выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы **`pMatrix`**. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора **`pResult`**.

7.7. Реализация параллельного алгоритма

При выполнении этого упражнения необходимо реализовать параллельный алгоритм матрично-векторного умножения на основе имеющейся реализации последовательного алгоритма. При этом будут рассмотрены следующие возможности библиотеки TBB:

- инициализация библиотеки;
- настройка проекта для работы с библиотекой;
- использование алгоритма **`tbb::parallel_for`**.

Кроме перечисленного выше будет изучено влияние значений параметров библиотеки на время выполнения вычислений.

Задание 1. Открытие проекта ParallelMatrixVectorMult

Откройте проект **ParallelMatrixVectorMult**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLabs\ParallelMatrix Vector**;
- дважды щелкните на файле **ParallelMatrixVectorMult.sln** или подсветите его выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **ParallelMV.cpp**, как это показано на рис. 6. После этих действий код, который вам предстоит модифицировать, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

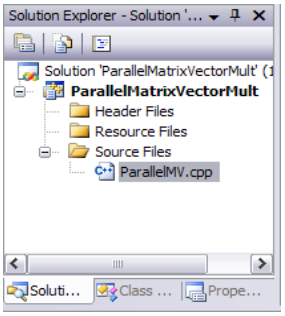

Рис. 6. Открытие файла ParallelMV.cpp

В файле **ParallelMV.cpp** расположена главная функция (**`main`**) будущего параллельного приложения, которая содержит объявления необходимых переменных. Также в файле **ParallelMV.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм умножения матрицы на вектор: **`DummyDataInitialization`**, **`RandomDataInitialization`**, **`SerialResultCalculation`**, **`PrintMatrix`** и **`PrintVector`**. Эти функции можно будет использовать и в параллельной программе.

Скомпилируйте и запустите приложение стандартными средствами **Microsoft Visual Studio 2005**. Убедитесь в том, что в командную консоль выводится приветствие: "`Parallel matrix-vector multiplication program`".

Задание 2. Настройка проекта для использования TBB

Первое, что нам предстоит изменить, – это подсказать **Microsoft Visual Studio 2005** место расположения заголовочных файлов библиотеки **TBB**. Эта задача может быть решена двумя способами:

- Запустить файл **C:\Program Files\Intel\<TBB Directory>\<arch>\vc8\bin\ tbbvars.bat**, где <arch> принимает значения ia32 (Intel® IA-32 processors) или em64t (Intel® EM64T processors).
- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать пункт **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Include files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **Include** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\include**. Нажать **OK**.

Следующий шаг – указание пути к статической библиотеке **TBB**, с которой будет собираться наше приложение. Таких библиотек две: **tbb_debug.lib** и **tbb.lib**. Первая библиотека выполняет различные проверки во время выполнения приложения и полностью поддерживается профилировщиком **Intel Thread Profiler**, предназначена для компиляции и сборки отладочных версий программ. Вторая библиотека имеет гораздо более эффективную реализацию функций и методов и предназначена для компиляции и сборки финальных версий. Разумеется, имеет смысл подключать одну из двух библиотек в зависимости от ситуации (отладка, финальная сборка).

Для подключения библиотеки необходимо выполнить следующую последовательность действий:

- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Library files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **lib** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\vc8\lib**. Нажать **OK**.
- В окне **Solution Explorer** нажать правой кнопкой мыши на названии проекта (**ParallelMatrixVectorMult**) и выбрать пункт **Properties**.
- Выбрать пункт **Linker\Input** и в поле **Additional Dependencies** ввести название библиотеки: **tbb_debug.lib** (для **Debug**-сборки) или **tbb.lib** (для **Release**-сборки).

Задание 3. Инициализация библиотеки TBB

Библиотека TBB является библиотекой классов, соответственно, вся (или почти вся) функциональность реализована посредством классов.

Началу работы с библиотекой предшествует обязательная процедура инициализации. Для этого необходимо создать экземпляр класса **task_scheduler_init**. Конструктор этого класса в зависимости от полученных параметров выполняет следующие действия:

- Указывает библиотеке TBB на необходимость автоматически определить число создаваемых потоков (**task_scheduler_init::automatic** является значением по умолчанию).
- Указывает библиотеке на необходимость создать нужное разработчику число потоков (параметр конструктора – число потоков).
- Указывает библиотеке на необходимость отложить инициализацию до момента, когда это будет необходимо разработчику (**task_scheduler_init::deferred**). В этом случае сама инициализация происходит только после вызова метода **task_scheduler_init::initialize** с параметром *n* – число потоков, которое необходимо разработчику.

При выполнении данной лабораторной работы мы рекомендуем придерживаться схемы, подразумевающей инициализацию планировщика потоков в начале программы с параметром по умолчанию (автоматическое определение числа потоков) и завершение работы библиотеки в конце программы. Достаточно часто предлагаемая схема обеспечивает наилучшую производительность.

Добавим в наш код инициализацию и завершение работы библиотеки. Для этого подключим заголовочный файл tbb/task_scheduler_init.h, выполним создание и уничтожение объекта – инициализатора библиотеки.

```
#include "tbb/task_scheduler_init.h"
<...>
void main()
{
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  // Result vector for matrix-vector multiplication
  double* pResult;
  int Size;         // Sizes of initial matrix and vector

  <...>

  task_scheduler_init init;

  <...>
  return 0;
}
```

Обратите внимание, что завершение работы библиотеки происходит при выходе из функции **main** в результате автоматического вызова деструктора локального объекта **init**.

Выполните компиляцию и сборку. Убедитесь, что сообщения об ошибках отсутствуют.

Задание 4. Распараллеливание цикла средствами библиотеки TBB

Разработаем параллельную программу для умножения матрицы на вектор при ленточной схеме разделения данных по строкам. Для организации параллельных вычислений предполагается использование многопроцессорных систем с общей памятью, что позволяет нам использовать TBB.

Параллельный алгоритм состоит в умножении строк матрицы на вектор с использованием нескольких потоков. Будем понимать далее под *итерацией* процедуру вычисления скалярного произведения строки матрицы на вектор. Тогда каждый поток выполняет несколько таких итераций, фактически, умножая горизонтальную полосу матрицы **pMatrix** на вектор **pVector** и вычисляя блок элементов результирующего вектора **pResult**. Заметим полное отсутствие зависимости по данным между разными итерациями, что создает предпосылки для хорошего распараллеливания.

Заметим также, что матрица **pMatrix** и векторы **pVector** и **pResult** являются общими для всех потоков. Однако элементы матрицы **pMatrix** и вектора **pVector** используются только на чтение, а элементы вектора **pResult** между потоками разделяются без пересечения. Тем самым, использование общих данных является корректным и организация параллельных вычислений не составляет больших проблем.

Итак, схема вычислений может быть представлена в виде цикла, итерацией которого является скалярное произведение. Библиотека TBB предоставляет возможность легко реализовать параллельную версию таких вычислений. Для этого в библиотеке имеется шаблонная функция **tbb::parallel_for**.

```
template<typename Range, typename Body>
void parallel_for(const Range& range, const Body& body);
```

Первый аргумент этой функции – **range** – итерационное пространство, к которому необходимо применить метод **body::operator()**. Второй аргумент функции – функтор.

Подробное описание того, что такое итерационное пространство, может быть найдено в разделе 6.5. Здесь же мы лишь напомним, что итерационное пространство в TBB определяет способ разделения общих данных между параллельно работающими потоками. Необходимое нам в этой задаче одномерное итерационное пространство (действительно, мы имеем **Size** итераций, состоящих в вычислении скалярного произведения) моделируется шаблонным классом **blocked_range**.

```
template<typename Value> class blocked_range;
```

Такое пространство задает диапазон значений в виде полуинтервала **[begin, end)**, с размером порции (количеством итераций), обрабатываемым одним потоком, равным **grainsize**.

В нашей задаче имеет смысл считать, что **begin = 0**, **end = Size**. Задав **grainsize = 1** (по умолчанию), мы реализуем максимально возможный параллелизм, так как библиотека сможет при наличии достаточного числа процессоров/ядер создать число потоков, равное **Size**. Отметим также, что в реальной ситуации (когда число процессоров/ядер много меньше **Size**) в отличие от OpenMP задание **grainsize = 1** не означает, что потоки будут обрабатывать итерации в каком бы то ни было заранее нам известном регулярном порядке. Решение о том, какая итерация каким потоком будет выполняться, принимает библиотека. Средств управления порядком выполнения в TBB нет. Более подробную информацию по данному вопросу см. в разделе 6.5.

Задание параметров выполняется при помощи следующего конструктора:

```
blocked_range(Value begin, Value end, size_t grainsize = 1);
```

О выборе оптимального значения параметра **grainsize** мы поговорим чуть позже, а сейчас перейдем к рассмотрению второго параметра функции **parallel_for** – функтора **body**.

В применении к **parallel_for** функтор определяет действие, которое должно быть выполнено потоком над пакетом итераций размером **grainsize**. Функтор для **parallel_for** должен удовлетворять следующим требованиям:

- в качестве полей класса иметь все общие для потоков данные;
- иметь конструктор-инициализатор;
- иметь конструктор копирования;
- иметь деструктор;
- иметь константный метод **operator()**, получающий константную ссылку на итерационное пространство и реализующий вычисления.

Функция **parallel_for** выполняет разделение итерационного пространства (**range**) на непересекающиеся подмножества размером **grainsize** и вызывает для каждого подмножества метод **body::operator()** в отдельном потоке. Заметим, что перед запуском вычислений **parallel_for** создает копию функтора для каждого потока, что требует корректно реализованного конструктора копирования. После завершения работы **parallel_for** уничтожает созданные объекты (что требует наличия деструктора).

Приступим к реализации функтора для решаемой задачи. Напомним, что функтор реализует итерации, то есть в рамках выбранной схемы вычислений подсчитывает скалярные произведения векторов.

```
class MatrixVectorMultiplicator
{
  const double *pMatrix,
               *pvector,
               *pResult;
  const int Size;

public:
  void operator()( const blocked_range<int>& r ) const
  {
    int begin = r.begin();
    int end   = r.end();

    for (int i = begin; i != end; i++)
    {
      pResult[i] = 0;
      for (int j = 0; j < Size; j++)
        pResult[i] += pMatrix[i * Size + j] * pVector[j];
    }
  }

  MatrixVectorMultiplicator(double *pm, double *pv, double *pr,
    int sz): pMatrix(pm), pVector(pv), pResult(pr), Size(sz)
  {}
};
```

Функтор содержит поля, в которых хранятся значения переменных, участвующих в вычислениях в методе **operator()**. Инициализация этих полей осуществляется с помощью конструктора. Конструктор копирования и деструктор, создаваемые компилятором по умолчанию, в данном случае корректны, поэтому их реализация в явном виде не приводится. Значения начальной и конечной итераций (**r.begin()**, **r.end()**) считаются вне тела цикла. Это сделано для того, чтобы эти функции не вызывались на каждой итерации, т. к. это может привести к замедлению работы приложения.

Реализуем функцию параллельного вычисления умножения матрицы на вектор при помощи функции **parallel_for** из библиотеки TBB:

```
void ParallelResultCalculation (double* pMatrix, double* pVector,
  double* pResult, int Size, int grainsize)
{
  parallel_for(blocked_range<int>(0, Size, grainsize),
    MatrixVectorMultiplicator(pMatrix, pVector, pResult, Size));
}
```

Параметр **grainsize** у разработанной функции предусмотрен для будущих экспериментов с размером порции итераций.

Добавим вызов функции параллельного умножения матрицы на вектор в функцию **main** (не забудем подключить еще 2 заголовочных файла). В качестве значения **grainsize** укажем размер вектора.

```
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

<...>

void main()
{
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector
                    // multiplication
  int Size;         // Sizes of initial matrix and vector

  <...>
  task_scheduler_init init;
  <...>

  ParallelResultCalculation(pMatrix, pVector, pResult,Size, Size);
  PrintVector(pResult, Size);

  <...>

  return 0;
}
```

Задание 5. Проведение вычислительных экспериментов

Выполните сборку и запуск приложения. Убедитесь в отсутствии ошибок и правильной работе приложения. Убедитесь, что время работы параллельного приложения осталось примерно таким же, как и у последовательного.

Задание 6. Выбор значения параметра grainsize

Параметр **grainsize** задает число итераций из итерационного пространства, которое составит размер порции вычислений для передачи отдельному потоку. Итерационное пространство распределяется между всеми потоками в зависимости от значения параметра **grainsize**. Принцип распределения описан в разделе 6.5 в пункте «Планирование вычислений».

Выбор значения параметра **grainsize** сильно влияет на производительность параллельной программы. При этом невозможно указать какое-то одно «замечательное» значение, которое подойдет для всех задач. На практике **grainsize** подбирается экспериментально.

Руководствуясь описанным в пункте «Планирование вычислений» раздела 6.5 алгоритмом, реализуем программный код для подбора значения параметра **grainsize**. Начальное значение **grainsize** возьмем равным **Size**. Далее, в цикле, будем уменьшать это значение в 2 раза, пока оно не станет равным 1. В зависимости от входных данных (**Size**) желаемое значение **grainsize** будет меняться.

```
int main()
{
  double *pMatrix = NULL,
         *pVector = NULL,
         *pResult = NULL;

  int Size = 0;

  task_scheduler_init init(1);

  printf("Parallel matrix-vector multiplication program\n");

  ProcessInitialization(pMatrix, pVector, pResult, Size);

  //Computing experiments
  const int repeatCount = 64;
  tick_count startTime;
  double duration;

  for (int grainsize = Size; grainsize >= 1; grainsize /= 2)
  {
    double minDuration = DBL_MAX;

    // Matrix-vector multiplication. Parallel version based on TBB
    // ...

    printf("Grain size is %d, execution time is %f s.\n",
            grainsize, minDuration);
  }

  ProcessTermination(pMatrix, pVector, pResult);
  getch();

  return 0;
}
```

Выполните сборку и запуск приложения. Определите, как меняется время работы при уменьшении **grainsize** (рис. 7). Определите желаемое значение **grainsize**.

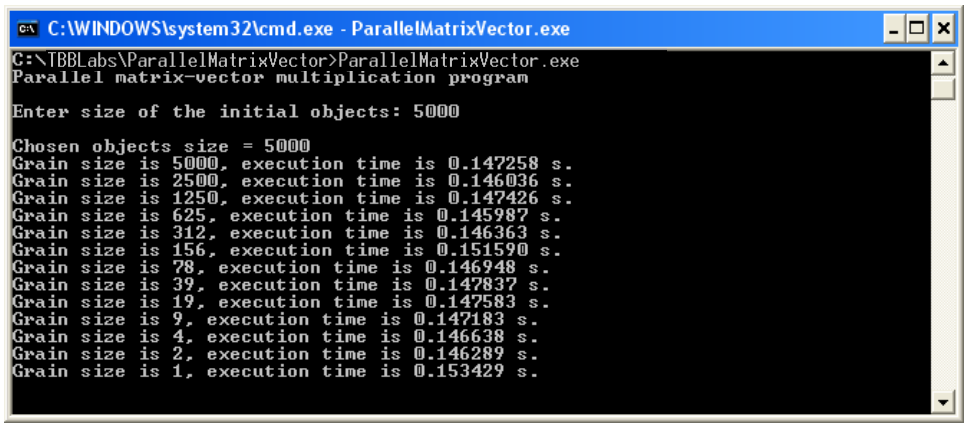

Рис. 7. Результаты эксперимента по подбору значения **grainsize**

Руководствуясь приведенным выше алгоритмом по подбору значения **grainsize** и результатами вычислительного эксперимента, представленными на рис. 7, можно сделать вывод, что даже минимальное значение **grainsize = 1** будет подходящим для задачи умножения квадратной матрицы на вектор.

Задание 7. Оценка эффективности

Выполните сборку приложения в конфигурации **Release**. Запустите программу несколько раз, задавая одни и те же размеры исходных данных. Как и ранее, мы рекомендуем выполнить приложение несколько раз и выбрать минимальное из полученных времен работы при одних и тех же размерах матрицы и вектора.

Заполните недостающие столбцы таблицы 2.

Таблица 2

Время работы параллельного приложения

Номер теста	Параметр Size	Время работы последовательного приложения (сек)	Время работы параллельного приложения (сек)	Выбранное значение **grainsize**
1	250			
2	500			
3	1000			
4	2000			
5	4000			
6	8000			
7	10000			

На основе значений, записанных в таблицу 2, посчитайте ускорение вычислений и заполните таблицу 3.

Таблица 3

Ускорение вычислений

Номер теста	Параметр Size	Ускорение параллельного алгоритма
1	250	
2	500	
3	1000	
4	2000	
5	4000	
6	8000	
7	10000	

Проведите вычислительные эксперименты для задачи с фиксированными значениями исходных данных (4000×4000) с различными значениями **`grainsize`** и заполните два первых столбца таблицы 4. Выберите желаемое значение **`grainsize`** и заполните последнее поле в таблице 4.

Таблица 4

Подбор значения параметра `grainsize`

Номер теста	Значение **`grainsize`**	Время работы последовательного приложения (сек)	Время работы параллельного приложения (сек)	Выбранное значение **`grainsize`**
1	4000			
2	1000			
3	250			
4	64			
5	16			
6	4			

7.8. Контрольные вопросы

1. Какие факторы влияют на время работы приложения?
2. Как собрать приложение, использующее библиотеку TBB?
3. Как инициализировать библиотеку TBB?
4. Как завершить работу библиотеки TBB?
5. Как инициализировать библиотеку TBB с определенным числом программных потоков?
6. Каким образом организовано распараллеливание цикла **`for`** в библиотеке TBB?
7. Что такое функтор?
8. Что такое итерационное пространство?
9. Каков смысл параметра **`grainsize`**?
10. Как работает конструкция **`parallel_for`** в библиотеке TBB?
11. Какие методы функтора необходимо реализовать при использовании алгоритма **`parallel_for`**?
12. Как выбрать значение параметра **`grainsize`**?

7.9. Задания для самостоятельной работы

1. Измените созданную реализацию на случай неквадратной матрицы.
2. Разработайте программу умножения квадратных матриц.
3. Разработайте программу умножения прямоугольных матриц.

7.10. Литература

Использованные источники

1. Гергель В.П., Лабутина А.А. Умножение матрицы на вектор // Материалы образовательного комплекса «Параллельное программирование на OpenMP». Нижний Новгород, 2007.
2. Сиднев А.А., Сысоев А.В., Мееров И.Б. Библиотека Intel Threading Building Blocks – краткое описание // Материалы образовательного комплекса «Технологии разработки параллельных программ». Нижний Новгород, 2007.
3. Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
4. Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.

Рекомендуемая литература

5. Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. Reading, MA: Addison-Wesley, 2000 (русский перевод: Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. М.: Издательский дом «Вильямс», 2003).
6. Quinn M.J. Parallel Programming in C with MPI and OpenMP. New York, NY: McGraw-Hill, 2004.
7. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
8. Гергель В.П. Теория и практика параллельных вычислений. М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
9. Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.

Дополнительная литература

10. Березин И.С., Жидков И.П. Методы вычислений. М.: Наука, 1966.
11. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. СПб.: Питер, 2006.
12. Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. М.: ДМК Пресс, 2006.
13. Павловская Т.А. C/C++. Программирование на языке высокого уровня. СПб.: Питер, 2003.

14. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows: Пер. с англ. 4-е изд. СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.

Информационные ресурсы сети Интернет

15. Страница библиотеки TBB на сайте корпорации Intel – [http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm].
16. Сайт сообщества пользователей TBB – [http://threadingbuildingblocks.org].
17. Сайт Лаборатории параллельных информационных технологий НИВЦ МГУ –[http://www.parallel.ru].
18. Официальный сайт OpenMP – [www.openmp.org].

7.11. Приложения

7.11.1. Программный код последовательного умножения матрицы на вектор

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <windows.h>
#include "tbb/tick_count.h"

using namespace tbb;

// Function that converts numbers form LongInt type to double type
double LiToDouble(LARGE_INTEGER x)
{
double result =
  ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
  return result;
}

// Function that gets the timestamp in seconds
double GetTime()
{
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency(&lpFrequency);
  QueryPerformanceCounter(&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount) / LiToDouble(lpFrequency);
}

// Function for simple definition of matrix and vector elements
void DummyDataInitialization(double* pMatrix, double* pVector,
  int Size)
{
  int i, j;  // Loop variables

  for (i = 0; i < Size; i++) {
    pVector[i] = 1;
    for (j = 0; j < Size; j++)
```

```
      pMatrix[i * Size + j] = i;
  }
}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector,
  int Size)
{
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i = 0; i < Size; i++) {
    pVector[i] = rand() / double(1000);
    for (j = 0; j < Size; j++)
      pMatrix[i * Size + j] = rand() / double(1000);
  }
}

// Function for memory allocation and definition of object's
// elements
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size)
{
  // Size of initial matrix and vector definition
  do
  {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d\n", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);
  // Memory allocation
  pMatrix = new double [Size * Size];
  pVector = new double [Size];
  pResult = new double [Size];
  // Definition of matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix(double* pMatrix, int RowCount, int ColCount)
{
  int i, j; // Loop variables
  for (i = 0; i < RowCount; i++)
  {
    for (j = 0; j < ColCount; j++)
      printf("%7.4f ", pMatrix[i * RowCount + j]);
    printf("\n");
  }
}

// Function for formatted vector output
void PrintVector(double* pVector, int Size)
{
  int i;
```

```
  for (i = 0; i < Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size)
{
  int i, j;  // Loop variables
  for (i = 0; i < Size; i++)
  {
    pResult[i] = 0;
    for (j = 0; j < Size; j++)
      pResult[i] += pMatrix[i * Size + j] * pVector[j];
  }
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector,
  double* pResult)
{
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}

void main()
{
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector
                    // multiplication
  int Size;         // Sizes of initial matrix and vector
  double Duration;
  tick_count Start, Finish;

  printf("Serial matrix-vector multiplication program\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Matrix-vector multiplication
  //Start = GetTime();
  Start = tick_count::now();
  ResultCalculation(pMatrix, pVector, pResult, Size);
  //Finish = GetTime();
  Finish = tick_count::now();
  //Duration = Finish - Start;
  Duration = (Finish - Start).seconds();

  // Printing the result vector
```

```
  printf("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the time spent by matrix-vector multiplication
  printf("\n Time of execution: %f\n", Duration);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
}
```

7.11.2. Программный код параллельного матрично-векторного умножения

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <time.h>

#include "tbb/tick_count.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

using namespace tbb;
using namespace std;

class DotProductCalc
{
  double *pMatrix,
         *pVector,
         *pResult;

  int Size;

public:
  void operator()(const blocked_range<int>& r) const
  {
    int begin = r.begin();
    int end   = r.end();

    for (int i = begin; i != end; i++)
    {
      pResult[i] = 0;
      for (int j = 0; j < Size; j++)
        pResult[i] += pMatrix[i * Size + j] * pVector[j];
    }
  }

  DotProductCalc(double *pm, double *pv, double *pr, int sz):
     pMatrix(pm), pVector(pv), pResult(pr), Size(sz)
  {}
};

// Function for simple definition of matrix and vector elements
void DummyDataInitialization(double* pMatrix, double* pVector,
```

```
  int Size)
{
  int i, j;  // Loop variables

  for (i = 0; i < Size; i++)
  {
    pVector[i] = 1;
    for (j = 0; j < Size; j++)
      pMatrix[i * Size + j] = i;
  }
}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector,
  int Size)
{
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i = 0; i < Size; i++)
  {
    pVector[i] = rand() / double(1000);
    for (j = 0; j < Size; j++)
      pMatrix[i * Size + j] = rand() / double(1000);
  }
}

// Function for memory allocation and definition of object's
// elements
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size)
{
  // Size of initial matrix and vector definition
  do
  {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d\n", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);
  // Memory allocation
  pMatrix = new double [Size * Size];
  pVector = new double [Size];
  pResult = new double [Size];
  // Definition of matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix(double* pMatrix, int RowCount, int ColCount)
{
  int i, j; // Loop variables
  for (i = 0; i < RowCount; i++)
  {
    for (j = 0; j < ColCount; j++)
```

```
      printf("%7.4f ", pMatrix[i * RowCount + j]);
    printf("\n");
  }
}

// Function for formatted vector output
void PrintVector(double* pVector, int Size)
{
  int i;
  for (i = 0; i < Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for parallel matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size, int grainsize)
{
  parallel_for(blocked_range<int>(0, Size, grainsize),
    DotProductCalc(pMatrix, pVector, pResult, Size));
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector,
  double* pResult)
{
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}

void main()
{
  double* pMatrix; // The first argument - initial matrix
  double* pVector; // The second argument - initial vector
  double* pResult; // Result vector for matrix-vector
                   // multiplication
  int Size;        // Sizes of initial matrix and vector
  double Duration;
  tick_count Start, Finish;
  task_scheduler_init init;

  printf("Prallel matrix-vector multiplication program\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Matrix-vector multiplication
  //Start = GetTime();
  Start = tick_count::now();
  ParallelResultCalculation(pMatrix, pVector, pResult, Size, 1);
  //Finish = GetTime();
```

```
  Finish = tick_count::now();
  //Duration = Finish - Start;
  Duration = (Finish - Start).seconds();

  // Printing the result vector
  printf("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the time spent by matrix-vector multiplication
  printf("\n Time of execution: %f\n", Duration);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
}
```

8. Лабораторная работа. Использование механизма логических задач библиотеки Intel Threading Building Blocks на примере вычисления быстрого преобразования Фурье

8.1. Введение

В настоящей лабораторной работе рассматривается библиотека Intel Threading Building Blocks (TBB) [11], позволяющая разрабатывать параллельные программы для систем с общей памятью. Идея, лежащая в основе библиотеки, – использование стандартного высокоуровневого языка C++ для быстрой разработки кросс-платформенных, хорошо масштабируемых параллельных приложений (см. подробнее в [1]). Кроме того, библиотека TBB предоставляет механизмы абстрагирования от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении прикладной задачи, что является достаточно актуальным.

Дальнейшее изучение принципов функционирования и вопросов эффективного использования TBB производится на примере вычисления быстрого преобразования Фурье (БПФ). БПФ, с одной стороны, широко используется в практических задачах, возникающих в теории автоматического регулирования и управления, в теории фильтрации, в задачах цифровой обработки сигналов и т.д., с другой – реализация эффективной параллельной версии вычисления БПФ представляет собой весьма непростую задачу.

В ходе выполнения работы предполагается, что слушатели имеют навыки разработки объектно-ориентированных программ на C++, а также владеют основами параллельного программирования в системах с общей памятью.

8.2. Методические рекомендации

8.2.1. Цели и задачи работы

Целью данной лабораторной работы является приобретение практических навыков использования механизма логических задач библиотеки TBB.

Данная цель предполагает решение следующих задач:

- изучение способов инициализации и завершения работы библиотеки TBB (см. также [1]);

- освоение функциональности библиотеки TBB, связанной с механизмом логических задач (см. также [1]);
- рассмотрение учебной задачи, направленной на демонстрацию использования механизма логических задач библиотеки TBB;
- самостоятельная разработка и отладка параллельных программ с использованием механизма логических задач TBB.

8.2.2. Структура работы

Лабораторная работа состоит из введения, четырех разделов, списка дополнительных заданий и списка литературы. Во введении обосновывается актуальность использования TBB в процессе разработки параллельных программ. В первом разделе даются методические рекомендации к лабораторной работе: формулируются цели и задачи, системные требования, рекомендации по проведению занятий. Во втором разделе приводятся теоретические сведения, необходимые для дальнейшей реализации последовательного и параллельных алгоритмов преобразования Фурье. В третьем разделе описывается реализация последовательной версии вычисления БПФ. В четвертом разделе проводится реализация двух параллельных версий вычисления БПФ: с использованием функции **`tbb::parallel_for`**, а также на основе механизма логических задач библиотеки TBB. В заключение приводятся задания для самостоятельной проработки, а также использованная и рекомендуемая литература.

8.2.3. Системные требования

Приведем системные требования библиотеки TBB для Windows-систем. Аналогичную информацию для систем на базе Linux и Mac OS можно найти на официальном сайте TBB [11].

Аппаратное обеспечение

Минимальные требования

- Intel® Pentium® 4 процессор.
- 512 Мб ОЗУ.
- 300 Мб дискового пространства.

Рекомендуемые требования

- Intel Pentium 4 процессор с поддержкой технологии Hyper-Threading (HT Technology) или Intel® Xeon® процессор.
- 1 Гб ОЗУ.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

Программное обеспечение

- Microsoft Windows XP Professional, Microsoft Windows Server 2003 или Microsoft Windows Vista.
- Intel® C++ Compiler 9.0 for Windows или старше.
- Microsoft Visual C++ 7.1 или старше.
- Microsoft Internet Explorer 6.0 или старше.
- Adobe Reader 6.0 или старше.

8.2.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется придерживаться следующей последовательности изучения материала:

1. Изучить, руководствуясь материалом из [1], способы инициализации и завершения работы библиотеки TBB.
2. Рассмотреть, руководствуясь материалом из [1], функциональность библиотеки TBB, связанную с использованием механизма логических задач.
3. Разобрать предложенные варианты решения задачи вычисления быстрого преобразования Фурье.
4. Перейти к выполнению дополнительных заданий, состоящих в самостоятельной разработке и отладке параллельных программ, использующих механизм логических задач библиотеки TBB.

8.3. Схема вычисления быстрого преобразования Фурье

8.3.1. Теоретическая справка

В основе преобразования Фурье[10] (ПФ) лежит простая, но исключительно плодотворная идея – почти любую периодическую функцию можно представить суммой отдельных гармонических составляющих (синусоид и косинусоид с различными амплитудами, периодами и, следовательно, частотами).

Неоспоримым достоинством ПФ является его гибкость – преобразование может использоваться как для непрерывных функций, так и для дискретных. В последнем случае оно называется дискретным преобразованием Фурье (ДПФ).

ПФ часто применяется при решении задач, возникающих в теории автоматического регулирования и управления, в теории фильтрации, ДПФ – в задачах цифровой обработки сигналов и других.

Вычисление ДПФ, чаще всего в виде БПФ, реализовано во многих математических библиотеках, например Intel MKL, Intel IPP. ПФ может быть обобщено на случай многомерных функций.

Непрерывное преобразование Фурье – преобразование, которое применяется к функции $x(t)$, заданной на интервале $(-\infty, +\infty)$. В результате получается функция $y(u)$:

$$y(u) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i u t} dt .$$

Также существует обратное преобразование, которое позволяет по образу $y(u)$ восстановить исходную функцию $x(t)$:

$$x(t) = \int_{-\infty}^{+\infty} y(u) e^{2\pi i u t} du .$$

[10] Жан Батист Жозеф Фурье (1768 – 1830). Родился в г. Осере (Оксер), Франция, в семье портного. Остался круглым сиротой в восьмилетнем возрасте. Прошел обучение в военной школе. В 1796 году возглавил кафедру математического анализа в знаменитой Политехнической школе. Автор труда «Аналитическая теория тепла», где впервые была систематически использована идея о представлении функций суммой разноамплитудных, но кратночастотных синусоид и косинусоид.

Образ $y(u)$ является комплексной функцией вещественного аргумента, также и $x(t)$ может принимать не только вещественные, но и комплексные значения.

С непрерывным преобразованием Фурье удобно работать в теории, но на практике обычно приходится иметь дело с дискретными данными, когда задано не аналитическое выражение преобразуемой функции, а лишь набор ее значений на некоторой сетке. В таком случае принимается допущение, что за пределами этой сетки функция равна нулю, а интеграл аппроксимируется интегральной суммой:

$$y(u) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i u t} dt = \sum_{k=0}^{n-1} x_k \Delta_k \exp\left(-2\pi i u t_k\right).$$

В случае равномерной сетки формула упрощается. Также на равномерной сетке обычно избавляются от шага, чтобы получить безразмерную формулу:

$$y_p \equiv \sum_{k=0}^{n-1} x_k \exp\left(-\frac{kp}{n} 2\pi i\right), \quad p = \overline{0, n-1}.$$

Обратное преобразование в таком случае будет иметь вид:

$$x_k \equiv \frac{1}{n} \sum_{p=0}^{n-1} y_p \exp\left(\frac{kp}{n} 2\pi i\right), \quad k = \overline{0, n-1}.$$

Определенное таким образом дискретное преобразование Фурье сохраняет практически все свойства непрерывного (разумеется, с учетом перехода к дискретному множеству).

Представленная выше формула для вычисления преобразования Фурье требует значительных затрат. Трудоемкость такого алгоритма имеет порядок $O(n^2)$. На практике вместо ДПФ используют быстрое преобразование Фурье (БПФ). БПФ – это простой алгоритм для эффективного вычисления ДПФ с трудоемкостью $O(n \log n)$. В настоящий момент существует несколько алгоритмов быстрого преобразования Фурье. В данной лабораторной работе рассматривается одна из наиболее популярных реализаций алгоритма, предложенная Кули и Таки (Cooley – Tukey) [4, 10].

8.3.2. Алгоритм Cooley – Tukey

Идея БПФ состоит в сокращении числа умножений и сложений, выполняемых в исходном ДПФ. Нетрудно заметить, что при вычислении ДПФ приходится многократно считать множители $e^{-\frac{jk}{n} 2\pi i}$. Например при всех $j = \overline{0, n-1}$ и $k = \overline{0, n-1}$ таких, что $(j \cdot k) \bmod n = const$, множители $e^{-\frac{jk}{n} 2\pi i}$ получаются одинаковыми. Очевидно, что таких j и k достаточно много. Например, $(j \cdot k) \bmod n = 0$ достигается при $j = 0$ и любом $k = \overline{0, n-1}$ или при $k = 0$ и любом $j = \overline{0, n-1}$. За счет использования этого факта и происходит экономия числа арифметических операций.

Алгоритм Cooley – Tukey разбивает исходное множество точек на два равных подмножества (размер входных данных должен быть степенью двойки). Каждое из подмножеств снова делится на 2 части и т.д. Над каждым из полученных множеств

выполняется БПФ. Вычисления можно выполнять независимо, этот факт будет использован при реализации параллельной версии.

Алгоритм Cooley – Tukey состоит из двух шагов:

1. Преобразование входного массива данных (*бит-реверсирование*).
2. Вычисление БПФ.

Бит-реверсирование

Данный этап заключается в изменении порядка следования исходных данных таким образом, чтобы с ними было удобно работать в дальнейшем. В некоторых алгоритмах БПФ этот этап пропускается и внедряется в вычислительный этап.

Бит-реверсирование – это преобразование двоичного числа путем изменения порядка следования бит в нем на противоположный. Бит-реверсирование применимо только к индексам элементов массива и предназначено для изменения порядка следования этих элементов, при этом значения самих элементов не изменяются.

Рассмотрим алгоритм работы на примере. Пусть исходный массив содержит 16 элементов, тогда преобразование индексов будет происходить следующим образом:

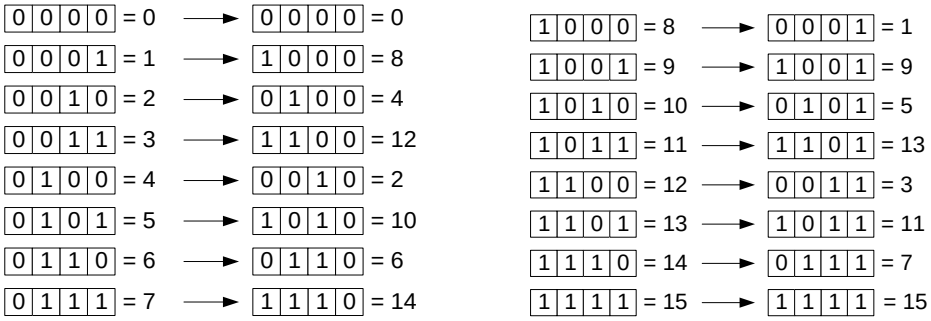

Рис. 1. Пример выполнения бит-реверсирования

Таким образом, исходный массив ($a[0], a[1], ..., a[15]$) после бит-реверсирования будет иметь вид: $a[0], a[8], a[4], a[12], a[2], a[10], a[6], a[14], a[1], a[9], a[5], a[13], a[3], a[11], a[7], a[15]$.

Программный код, выполняющий бит-реверсирование, представлен ниже.

```
void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
```

```
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}
```

Вычисление БПФ

Рассмотрим алгоритм вычисления БПФ на примере. Пусть на вход подается массив из 4 элементов (x_0, x_1, x_2, x_3). Выполнив бит-реверсирование, получим следующий порядок элементов (x_0, x_2, x_1, x_3). Собственно вычисления проиллюстрируем на рисунке 2.

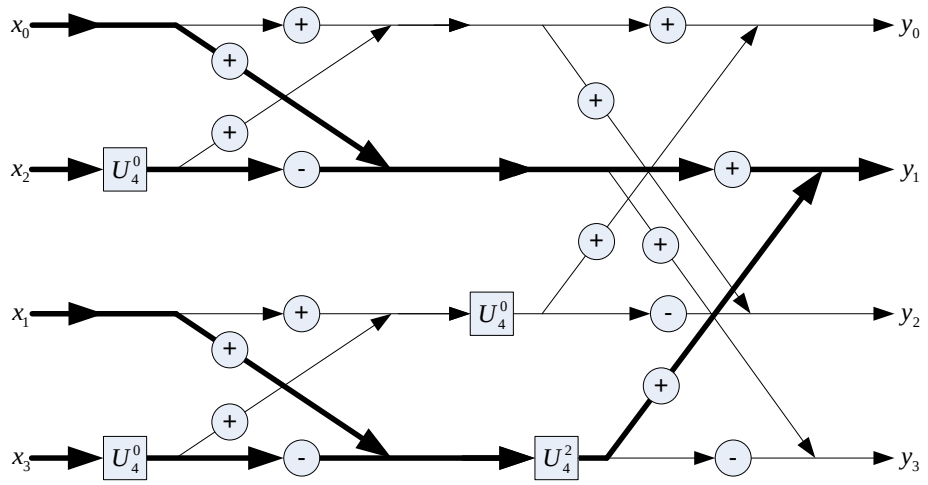

Рис. 2. Пример вычисления БПФ

Поскольку размер входного массива равен 4, вычисления состоят из двух этапов. В общем случае, когда размер входных данных равен степени двойки ($n = 2^m$), число этапов вычисления БПФ равно $\log n = m$. На каждом этапе многократно выполняется одна и та же базовая операция, которую часто называют «бабочкой». Вход «бабочки» представляют собой два числа: a и b. Выход – два других числа: $a + bU$ и $a - bU$. Число U называется *коэффициентом поворота*. На рис. 3 представлено графическое изображение указанной операции, из него читателю должно быть ясно происхождение ее названия.

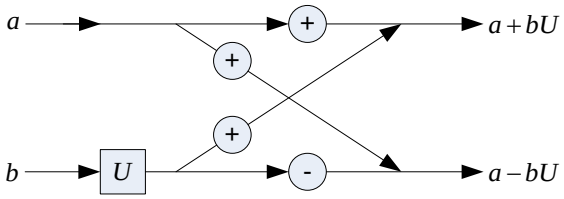

Рис. 3. Базовая операция вычисления БПФ «бабочка»

Итак, на первом этапе вычислений происходит применение «бабочки» с шагом 1 ко всем элементам, т.е. ко всем парам, состоящим из соседних элементов входного массива. В нашем примере это пары (x_0, x_2) и (x_1, x_3) (рис. 4). Общая формула коэффициента поворота $U_{bSize}^{j} = e^{-\frac{j}{bSize}\pi i}$, где *bSize* – размер шага «бабочки», j – номер итерации вычислений над группой элементов, размером 2 * *bSize* ($0 \le j < bSize$). На первом этапе вычислений коэффициент поворота «бабочки» будет равен 1 ($U_1^0 = e^0 = 1$) для всех обрабатываемых пар.

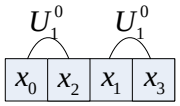

Рис. 4. 1-й этап вычисления БПФ над массивом из 4 элементов

На втором этапе вычислений шаг «бабочки» увеличится в два раза, т.е. будет равен 2. Таким образом «бабочка» будет применяться к следующим парам элементов (x_0, x_1) и (x_2, x_3) (рис. 5). При этом коэффициенты поворота для этих элементов будут различны: $U_2^0 = e^0 = 1$ и $U_2^1 = e^{-\frac{\pi}{2}i} = -i$ соответственно.

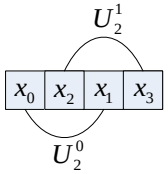

Рис. 5. 2-й этап вычисления БПФ над массивом из 4 элементов

Таким образом, для вычисления БПФ над массивом из 4 элементов потребуется 4 раза выполнить «бабочку». Как видно из рис. 2 (жирные стрелки), на выходе y_1 будут присутствовать все элементы, подаваемые на вход, с разными коэффициентами (за счет разных коэффициентов поворота). Можно убедиться, что полученное значение y_1 полностью совпадает со значением, полученным при вычислении обычного ДПФ ($y_1 = \sum_{j=0}^{3} x_j e^{-\frac{j}{2}\pi i} = x_0 - ix_1 - x_2 + ix_3$). Аналогичным образом обстоит ситуация с остальными выходами.

Рассмотрим общий алгоритм вычисления БПФ. Пусть $n = 2^m$ – размер входных данных, **`signal`** – входной массив комплексных чисел.

1. **`bSize = 1`** (шаг «бабочки»).
2. **`i = 0, j = 0.`**
3. Применить «бабочку» к элементам **`signal[i * bSize * 2 + j]`** и **`signal[i * bSize * 2 + j + bSize]`** с коэффициентом поворота. $U = e^{-\frac{j}{bSize}\pi i}$
4. Если **`j < bSize - 1`**, то **`j++`**, переход на шаг 3.
5. Если **`i < bSize * n / 2 - 1`**, то **`i++, j = 0`**, переход на шаг 3.
6. Если **`bSize < n/2`**, то **`bSize = bSize * 2`**, переход на шаг 2, иначе завершение алгоритма.

8.4. Разработка последовательного приложения

Проведем пошагово реализацию последовательной программы вычисления БПФ. Заметим, что вариант, который будет рассмотрен, разумеется, не является единственным. Будем также считать основной целью корректность работы полученной реализации, а не ее производительность на какой-либо конкретной архитектуре.

Исходными данными для вычисления БПФ является входной сигнал, заданный в виде массива комплексных чисел. Входной сигнал будет задаваться 2 способами:

- специальный вид входного сигнала (для оценки корректности реализации алгоритма);
- случайная генерация данных (для проведения вычислительных экспериментов).

Для начала работы нам потребуется простейший проект, на основе которого будет выполняться разработка приложения.

Задание 1. Открытие проекта SerialNonrecursiveFFT

Откройте проект **SerialNonrecursiveFFT**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**…;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLabs\SerialNonrecursiveFFT**;
- дважды щелкните на файле **SerialNonrecursiveFFT.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **SerialNonrecursiveFFT.cpp**, как это показано на рис. 6. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

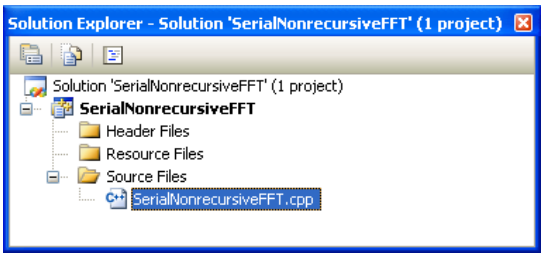

Рис. 6. Открытие файла SerialNonrecursiveFFT.cpp

В файле **SerialNonrecursiveFFT.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции **`main`**. Эта заготовка содержит объявление переменных и вывод на печать начального сообщения программы.

Быстрое преобразование Фурье, в общем случае, осуществляется над массивом комплексных чисел. Используем для работы с ними класс **`complex<double>`** библиотеки STL, подключив соответствующий заголовочный файл: **`#include <complex>`**. В функции **`main`** объявлены:

- **`inputSignal`** – входной сигнал, над которым будет выполняться БПФ, массив комплексных чисел **`complex<double>`**;
- **`outputSignal`** – выходной сигнал после применения БПФ, массив комплексных чисел **`complex<double>`**;
- **`size`** – размер входного и выходного сигнала (число элементов в сигнале в процессе преобразования Фурье не меняется), задает число элементов массивов **`inputSignal`** и **`outputSignal`**.

После объявления и инициализации начальными значениями переменных выводится начальное сообщение. Перед завершением приложения добавлено ожидание ввода с клавиатуры.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  cin.get();

  return 0;
}
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна **Microsoft Visual Studio 2005** появилось сообщение "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

После запуска исполняемого модуля в командной консоли появится сообщение: "Fast Fourier Transform. Serial version.", после чего программа будет ожидать нажатия клавиши **Enter**.

Задание 2. Ввод и генерация исходных данных

Для задания исходных данных последовательного алгоритма вычисления БПФ реализуем функцию **ProcessInitialization**. Эта функция предназначена для определения размера обрабатываемых данных (сигналов), выделения памяти под сигналы: входной **inputSignal** и выходной **outputSignal**, а также для задания значений элементов входного сигнала. Таким образом, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size);
```

На первом этапе необходимо определить размеры объектов (задать значение переменной **size**). С учетом особенностей схемы вычисления БПФ значение **size** должно быть степенью двойки. Для упрощения реализации будем полагать, что **size >= 4**. В тело функции **ProcessInitialization** добавьте выделенный фрагмент кода:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if (size < 4)
      cout << "Input signal length should be >= 4" << endl;
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two"
               << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;
}
```

В функцию **main** добавим вызов функции **ProcessInitialization**:

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  cin.get();

  return 0;
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что значение переменной **size** задается корректно.

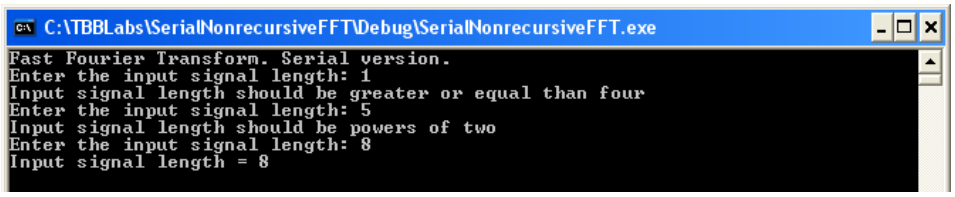

Рис. 7. Задание размера сигналов

Функция инициализации должна также выделять память для хранения объектов (добавьте выделенный код в тело функции **ProcessInitialization**):

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    <...>
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];
}
```

Далее необходимо задать значения элементов входного сигнала **inputSignal**. Для выполнения этих действий реализуем еще одну функцию **DummyDataInitialization**. Заполним массив входных данных нулями, кроме одного элемента **mas[size - size / 4] = 1.** Результатом применения БПФ к такому массиву будет массив из комплексных чисел вида $\cos(\frac{\pi}{2}k)+i\sin(\frac{\pi}{2}k)$,

$k = \overline{0, size-1}$. Используя такие начальные данные, легко визуально проверить корректность работы реализованного алгоритма при небольших входных значениях.

Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[size - size / 4] = 1;
}
```

Вызов функции **DummyDataInitialization** необходимо выполнить после выделения памяти внутри функции **ProcessInitialization**:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    <...>
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}
```

Задание 3. Завершение процесса вычислений

Перед собственно вычислением БПФ сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию **ProcessTermination**. Память выделялась для хранения входного **inputSignal** и выходного **outputSignal** сигналов. Следовательно, эти объекты необходимо передать в функцию **ProcessTermination** в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;
  delete [] outputSignal;
  outputSignal = NULL;
}
```

Вызов функции **ProcessTermination** необходимо добавить в самый конец главной функции **main**:

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

Задание 4. Реализация БПФ

Выполним теперь разработку основной вычислительной части программы. Для вычисления БПФ реализуем функцию **SerialFFT**, которая принимает на вход исходный сигнал **inputSignal** размера **Size**, а также указатель на место в памяти, где должен быть сохранен выходной сигнал **outputSignal**.

В соответствии с алгоритмом вычисления БПФ, изложенным в пункте 8.3.2, код этой функции должен быть следующий:

```
// FFT computation
void SerialFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  BitReversing(inputSignal, outputSignal, size);
  SerialFFTCalculation(outputSignal, size);
}
```

Здесь функция **BitReversing** реализует первый шаг вычисления БПФ: битреверсирование (ее код был приведен в пункте 8.3.2.1), а функция **SerialFFTCalculation** – второй шаг, собственно вычисление.

В пункте 8.3.2.2 был, в общем виде, рассмотрен алгоритм второго шага в вычислении БПФ. Приведем теперь его реализацию.

```
// FFT computation
void SerialFFTCalculation(complex<double> *signal, int size)
{
  int m = 0;
  for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++)
    //size = 2^m
    ;
  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
```

```
    int butterflySize = butterflyOffset >> 1;
    double coeff = PI / butterflySize;

    for (int i = 0; i < size / butterflyOffset; i++)
      for (int j = 0; j < butterflySize; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j + i * butterflyOffset,
          butterflySize);
  }
}
```

Здесь функция **Butterfly** реализует вычисление базовой операции алгоритма – «бабочки». На рис. 8 представлена схема вычисления «бабочки» с произвольным шагом (**butterflySize**) и произвольным начальным элементом (**offset**).

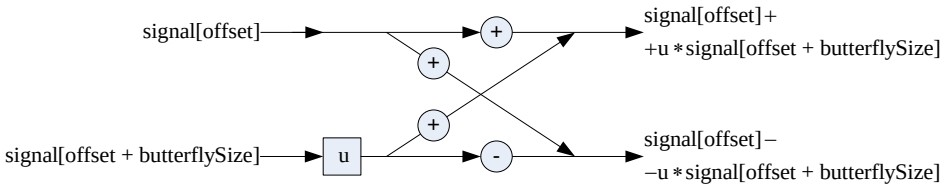

Рис. 8. Реализация «бабочки» в общем случае

Здесь **u** – комплексный коэффициент поворота, **signal** – массив комплексных чисел, над элементами которого вычисляется «бабочка». Таким образом, реализация функции **Butterfly** будет выглядеть так:

```
__inline void Butterfly(complex<double> *signal, complex<double> &u,
  int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}
```

Выполним вызов функции вычисления БПФ из основной программы.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // FFT computation
  SerialFFT(inputSignal, outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);
```

```
  cin.get();

  return 0;
}
```

Выполните сборку и запуск приложения. Убедитесь в отсутствии ошибок и правильной работе приложения. Вывод на консоль не изменился по сравнению с предыдущей сборкой, поэтому содержимое консоли после выполнения приложения не изменится. Не указывайте большой размер входного сигнала, т.к. это может привести к долгой работе приложения.

Задание 5. Проверка корректности

Необходимо определить правильность работы приложения. Для этого подадим на вход вычислительному алгоритму (**SerialFFT**) данные специального вида (сгенерированные с помощью функции **DummyDataInitialization**). Например, если размер входного сигнала равен 8, то после выполнения БПФ над сгенерированным массивом получим результат в виде массива из восьми комплексных чисел вида $\cos(\frac{\pi}{2}k)+i\sin(\frac{\pi}{2}k)$, $k=\overline{0,7}$.

Для отображения результатов работы алгоритма реализуем функцию вывода на экран значений массива комплексных чисел (**PrintSignal**).

```
void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}
```

Далее выведем значения результирующего сигнала и сравним их со значениями, посчитанными вручную.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // FFT computation
  SerialFFT(inputSignal, outputSignal, size);

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);
```

```
  cin.get();

  return 0;
}
```

Выполните сборку и запуск приложения. Проведите эксперименты с различными значениями размера входного сигнала. Убедитесь, что выводимые значения совпадают со значениями, посчитанными вручную. При вычислении БПФ могут возникнуть погрешности машинной арифметики, поэтому значения, посчитанные компьютером, могут не совпадать в точности с результатами, посчитанными теоретически.

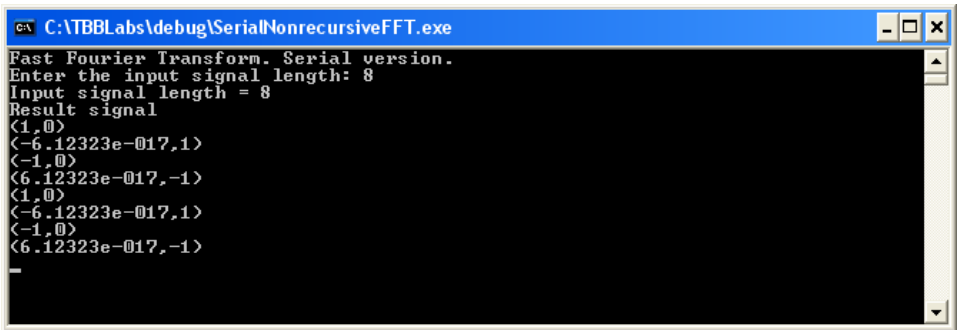

Рис. 9. Проверка корректности работы последовательной версии

Задание 6. Проведение вычислительных экспериментов

После реализации параллельной версии алгоритма нам потребуется оценивать ее ускорение. Для этого сейчас необходимо провести вычислительные эксперименты и замерить времена работы последовательной версии. Анализировать время выполнения последовательной реализации разумно для достаточно больших объектов. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов **`RandomDataInitialization`** (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}
```

Заметим, что в отладочных целях следует инициализировать датчик случайных чисел при помощи функции **`srand()`** одним и тем же числом, что обеспечит нам одинаковые последовательности случайных чисел и позволит обнаружить ошибки, если они есть.

Будем вызывать эту функцию вместо ранее разработанной функции **`DummyDataInitialization`**, которая генерировала данные так, чтобы можно было легко проверить правильность работы алгоритма:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
```

```
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    <...>
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  // Initialization of input signal elements - tests
  //DummyDataInitialization(inputSignal, size);

  // Initialization of input signal elements - computational experiments
  RandomDataInitialization(inputSignal, size);
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Замеры времени вычислений будем проводить при помощи класса **tick_count**. Чтобы оценки времени были корректны, будем проводить несколько экспериментов и выбирать наименьшее из времен, аналогично тому, как это делалось в лабораторной работе «Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения».

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  for(int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    // FFT computation
    SerialFFT(inputSignal, outputSignal, size);
    duration  = (tick_count::now() - startTime).seconds();

    if(duration < minDuration)
      minDuration = duration;
  }
```

```
  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими входными сигналами отключите печать векторов (закомментируйте соответствующие строки кода).

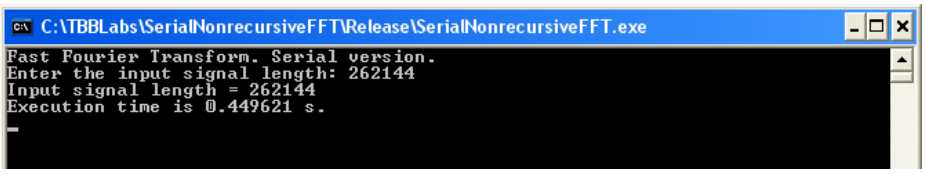

Рис. 10. Замеры времени работы последовательной версии

Задание 7. Оценка эффективности

Выполните сборку приложения в конфигурации **Release**. Проведите вычислительные эксперименты, результаты занесите в таблицу.

Таблица 1

Результаты экспериментов вычисления БПФ

Номер теста	Размер входного сигнала	Время работы последовательного приложения (сек)
1	32768	
2	65536	
3	131072	
4	262144	
5	524288	

8.5. Разработка параллельного приложения

В данном разделе будет рассмотрен процесс разработки параллельного приложения с использованием библиотеки TBB на основе последовательного приложения:

- создание параллельной версии с использованием алгоритма **tbb::parallel_for**;
- создание параллельной версии с использованием логических задач (класса **tbb::task**).

Задание 1. Открытие проекта ParallelNonrecursiveFFT

За основу разработки параллельного приложения возьмем последовательную версию. Функции инициализации, завершения, генерации данных, вывода на экран

результирующего сигнала, бит-реверсирования и вычисления «бабочки» останутся без изменений.

Откройте проект **ParallelNonrecursiveFFT**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**…;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLab\ParallelNonrecursiveFFT**,
- дважды щелкните на файле **ParallelNonrecursiveFFT.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **ParallelNonrecursiveFFT.cpp**, как это показано на рис. 11. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

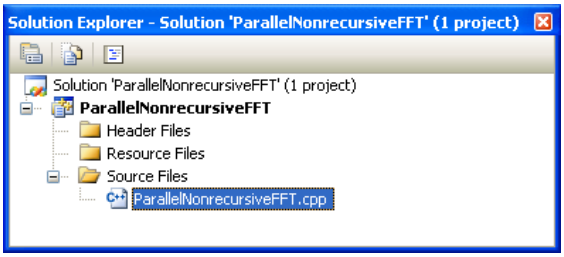

Рис. 11. Структура проекта ParallelNonrecursiveFFT

Соберите исполняемый модуль. Убедитесь, что компиляция прошла успешно. После запуска исполняемого модуля в командной консоли появится сообщение: `"Fast Fourier Transform. Parallel version."`, после чего программа будет ожидать нажатия клавиши **Enter**.

Задание 2. Настройка проекта для использования TBB

Первое, что нам предстоит изменить, – это подсказать **Microsoft Visual Studio 2005** место расположения заголовочных файлов библиотеки **TBB**. Эта задача может быть решена двумя способами:

- Запустить файл **C:\Program Files\Intel\<TBB Directory>\<arch>\vc8\bin\ tbbvars.bat**, где <arch> принимает значения ia32 (Intel® IA-32 processors) или em64t (Intel® EM64T processors).
- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать пункт **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Include files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **Include** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\include**. Нажать **OK**.

Следующий шаг – указание пути к статической библиотеке **TBB**, с которой будет собираться наше приложение. Таких библиотек две: **tbb_debug.lib** и **tbb.lib**. Первая библиотека выполняет различные проверки во время выполнения приложения и полностью поддерживается профилировщиком **Intel Thread Profiler**, предназначена для компиляции и сборки отладочных версий программ. Вторая

библиотека имеет гораздо более эффективную реализацию функций и методов и предназначена для компиляции и сборки финальных версий. Разумеется, имеет смысл подключать одну из двух библиотек в зависимости от ситуации (отладка, финальная сборка).

Для подключения библиотеки необходимо выполнить следующую последовательность действий:

- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Library files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **lib** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\vc8\lib**. Нажать **OK**.
- В окне **Solution Explorer** нажать правой кнопкой мыши на названии проекта (**ParallelNonrecursiveFFT**) и выбрать пункт **Properties**.
- Выбрать пункт **Linker\Input** и в поле **Additional Dependencies** ввести название библиотеки: **tbb_debug.lib** (для **Debug**-сборки), или **tbb.lib** (для **Release**-сборки).

Задание 3. Инициализация библиотеки TBB

Аналогично тому, как это делалось в лабораторной работе «Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения», для начала работы с библиотекой TBB необходимо создать хотя бы один экземпляр класса **`task_scheduler_init`**. Создадим объект этого класса без параметров (число потоков будет определено автоматически).

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;
<...>

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Задание 4. Разработка параллельной версии с использованием алгоритма tbb::parallel_for

Выполнив эксперименты, нетрудно определить, что бит-реверсирование занимает мало времени по сравнению с остальными вычислениями, поэтому распараллеливать имеет смысл функцию **SerialFFTCalculation**.

```
// FFT computation
void SerialFFTCalculation(complex<double> *signal, int size)
{
  <...>
  for (int p = 0; p < m; p++)
  {
    <...>

    for (int i = 0; i < size / butterflyOffset; i++)
      for (int j = 0; j < butterflySize; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j + i * butterflyOffset,
          butterflySize);
  }
}
```

Выделенный цикл в приведенном выше коде не имеет зависимостей по данным. Каждая итерация этого цикла может быть выполнена независимо, поэтому его можно распараллелить. На основе этого цикла реализуем функтор. Полями функтора будут все общие данные, использующиеся в вычислениях: **signal**, **butterflySize**, **coeff**. Поскольку **butterflyOffset = 2 * butterflySize**, переменную **butterflyOffset** не будем делать общей. Конструктор содержит только два аргумента, т.к. переменную **coeff** можно считать программно в теле конструктора.

```
class ButterflyCalculator
{
  complex<double> *const signal;
  int const butterflySize;
  double coeff;

public:
   void operator()( const blocked_range<int>& r ) const
   {
     int begin = r.begin(),
         end = r.end(),
         butterflyOffset = 2 * butterflySize;

     for (int i = begin; i != end; i++)
       for (int j = 0; j < butterflySize; j++)
         Butterfly(signal, complex<double>(cos(-j * coeff),
           sin(-j * coeff)), j + i * butterflyOffset,
           butterflySize);
  }

  ButterflyCalculator(complex<double> *tsignal,
```

```
    int tbutterflySize):
    signal(tsignal), butterflySize(tbutterflySize)
  {
    coeff = PI / butterflySize;
  }
};
```

Функтор содержит поля, в которых хранятся значения переменных, участвующих в вычислениях в методе **operator()**. Инициализация этих полей осуществляется с помощью конструктора. Конструктор копирования и деструктор, создаваемые компилятором по умолчанию, в данном случае корректны, поэтому их реализация в явном виде не приводится. Значения начальной и конечной итераций (**r.begin()**, **r.end()**) считаются вне тела цикла. Это сделано для того, чтобы эти функции не вызывались на каждой итерации, т. к. это может привести к замедлению работы приложения.

Реализуем функции параллельного вычисления БПФ при помощи функции **parallel_for** из библиотеки TBB:

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  int m = 0;
  for(int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++)
    //size = 2^m
    ;

  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    parallel_for(blocked_range<int>(0, size / butterflyOffset,
      grainsize), ButterflyCalculator(signal, butterflySize));
  }
}

void ParallelFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size, int grainsize)
{
  BitReversing(inputSignal, outputSignal, size);
  ParallelFFTCalculation(outputSignal, size, grainsize);
}
```

Параметр **grainsize** у разработанной функции предусмотрен для будущих экспериментов с размером порции итераций.

Добавим вызов **ParallelFFT** в функцию **main**, также добавим код, необходимый для проведения экспериментов с различными значениями **grainsize**.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;
```

```
  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int testCount = 17;
  const int repeatCount = 16;
  tick_count startTime;
  double duration;

  cout << setprecision(6);
  for(int j = 0; j < testCount; j++)
  {
    int grainsize = size >> j;
    if (grainsize < 1)
      break;

    double minDuration = DBL_MAX;

    for(int i = 0; i < repeatCount; i++)
    {
      startTime = tick_count::now();
      ParallelFFT(inputSignal, outputSignal, size, grainsize);
      duration = (tick_count::now() - startTime).seconds();

      if (duration < minDuration)
        minDuration = duration;
    }
    cout << "Grain size is " << grainsize << ", execution time is ";
    cout << minDuration << " s. " << endl;
  }

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Соберите исполняемый модуль. Убедитесь, что компиляция прошла успешно. После завершения работы приложения в командной консоли появится соответствующая информация (рис. 12)[11].

[11] Времена выполнения могут отличаться от указанных на рисунке.

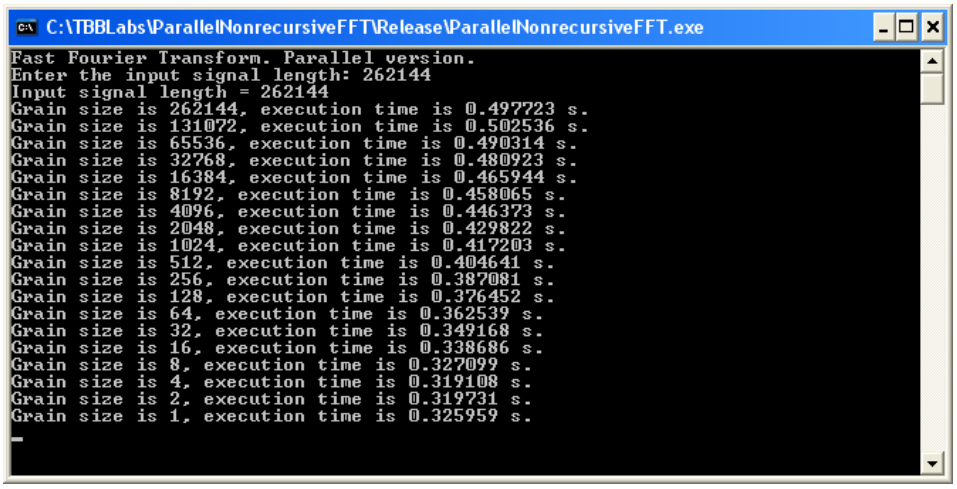

Рис. 12. Замеры времени параллельной версии при различных значениях **grainsize**

Выполните сборку приложения в конфигурации **Release**. Проведите эксперименты, задавая разные значения параметра **grainsize**, чтобы определить минимальное время, и заполните таблицу 2.

Таблица 2

Результаты экспериментов вычисления БПФ

Номер теста	Размер входного сигнала	Минимальное время работы параллельного приложения (сек)	Выбранное значение **grainsize**
1	32768		
2	65536		
3	131072		
4	262144		
5	524288		

Проведя эксперименты и заполнив таблицу, можно обнаружить, что для каждого размера входного сигнала оптимальные значения **grainsize** очень малы. Это связано с тем, что в функции **ParallelFFTCalculation** итерационное пространство алгоритма **parallel_for** меняется в зависимости от номера итерации внешнего цикла. Таким образом, объем вычислений на каждой итерации экспоненциально уменьшается, а **grainsize** остается постоянным. Это означает, что с некоторой итерации внешнего цикла размер итерационного пространства станет меньше, чем значение **grainsize**, или равен ему, поэтому приложение, начиная с этого шага, будет работать последовательно. Оптимальное значение **grainsize** с этой точки зрения равно 1, но оно не будет оптимальным с точки зрения производительности. Поэтому **grainsize** нужно пересчитывать на каждой итерации внешнего цикла.

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  int m = 0;
```

```
  for(int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++)
    //size = 2^m
    ;

  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1 ;

    int grain = grainsize / butterflyOffset;
    if (grain < 1)
      grain = 1;

    parallel_for(blocked_range<int>(0, size/butterflyOffset,
      grain), ButterflyCalculator(signal, butterflySize));
  }
}
```

Выполните сборку приложения в конфигурации **Release**. Проведите эксперименты и заполните таблицу 3.

Таблица 3

Результаты экспериментов вычисления БПФ

Номер теста	Размер входного сигнала	Минимальное время работы параллельного приложения (сек)	Выбранное значение **grainsize**
1	32768		
2	65536		
3	131072		
4	262144		
5	524288		

Задание 5. Разработка параллельной версии с использованием логических задач

До сих пор мы рассматривали нерекурсивную реализацию вычисления БПФ. Основным соображением при этом было то, что рекурсия довольно плохо поддается распараллеливанию, например реализовать параллельную рекурсивную программу на OpenMP – задача совершенно нетривиальная. Однако нетрудно заметить, что сам алгоритм вычисления БПФ целиком построен на рекурсии и, конечно, обычно именно через нее и реализуется. Вот как будет выглядеть типичная его реализация:

```
// FFT computation
void SerialFFTCalculation(complex<double> *const signal, int size)
{
  if (size == 1)
    return;

  double const coeff = 2.0 * PI / size;

  SerialFFTCalculation(signal, size / 2);
  SerialFFTCalculation(&(signal[size / 2]), size / 2);

  for (int j = 0; j < size / 2; j++)
```

```
    Butterfly(signal, complex<double>(cos(-j * coeff),
      sin(-j * coeff)), j , size / 2);
}
```

Как видим, код существенно упростился по сравнению с нерекурсивной версией. Вычисление БПФ от входного сигнала есть вычисление БПФ от каждой из его половин и т.д.

Библиотека TBB содержит средства, позволяющие эффективно, а главное – достаточно просто распараллеливать рекурсивные алгоритмы. Это средство – *логические задачи*. Подробное описание логических задач и их использования приведено в разделе 6.7. Здесь же мы отметим минимально необходимые моменты.

Логическая задача представлена в TBB в виде класса **tbb::task.** Он является базовым при реализации задач, т.е. этот класс должен быть унаследован всеми пользовательскими задачами.

Класс **tbb::task** содержит виртуальный метод **execute**, в котором и происходят все вычисления. Его прототип:

```
virtual task* execute();
```

В этом методе производятся необходимые вычисления, и после этого возвращается указатель на следующую задачу, которую необходимо выполнить. Если возвращается **NULL**, то вычисления прекращаются.

Создание задачи осуществляется с помощью оператора **new**. Некоторые его виды представлены ниже:

- **new(task::allocate_root()) T** – создание «главной» задачи типа **T**;
- **new(this.allocate_child()) T** – создание подзадачи типа **T.**

Может потребоваться синхронизация выполнения задач между собой. Для этих целей существует метод **void wait_for_all()**. Задача, для которой вызван этот метод, будет ожидать завершения всех своих подзадач. Перед тем как вызвать этот метод, необходимо с помощью метода **void set_ref_count(int count)** указать число подзадач для данной задачи + 1 (**set_ref_count(n + 1)**, где **n** – число подзадач для данной задачи).

После создания задач их необходимо запустить. Для этого существуют следующие методы:

- **void spawn(task& child)** – помещает задачу **child** в пул готовых к выполнению. Неблокирующая функция.
- **void spawn_and_wait_for_all(task& child)** – помещает задачу **child** в пул готовых к выполнению и ожидает завершения всех подзадач. Алгоритм работы аналогичен последовательному вызову методов **spawn** и **wait_for_all**.
- **static void spawn_root_and_wait(task& root)** – запускает задачу **root** на выполнение. Память для этой задачи должна быть выделена с помощью **task::allocate_root().**

Реализуем класс логической задачи для вычисления БПФ. Как и в случае функтора, полями класса будут все общие данные, использующиеся в вычислениях: **signal**, **butterflySize**, **coeff.**

Вычисления последовательной версии осуществляются рекурсивно. При этом на каждом шаге происходит разбиение задачи на две равные подзадачи вычисления БПФ. Используем этот факт. Вместо рекурсивного вызова будем создавать две подзадачи и выполнять их параллельно. При этом после завершения этих задач

необходимо выполнить один этап БПФ последовательно. В том случае когда размер подзадач станет невелик, будем вычислять их без дальнейшего подразбиения на задачи (последовательно).

```
class FFTTask: public task
{
  complex<double> *const signal;
  int const size;
  double coeff;

public:
  FFTTask(complex<double> *const tSignal, int tSize):
    signal(tSignal), size(tSize)
  {
    coeff = 2.0 * PI / size;
  }

  task* execute()
  {
    if (size < 1024)
      SerialFFTCalculation(signal, size);
    else
    {
      FFTTask& a =
        *new (allocate_child()) FFTTask(signal, size / 2);
      FFTTask& b =
        *new (allocate_child()) FFTTask(&(signal[size / 2]),
        size/2);

      set_ref_count(3);

      spawn(b);
      spawn_and_wait_for_all(a);

      for (int j = 0; j < size / 2; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j , size / 2);
    }

    return NULL;
  }
};
```

Как видим, рекурсия естественным образом преобразовалась в создание подзадач для половинок пришедшего в текущую задачу входного сигнала.

Создание и запуск «главной» задачи поместим в функцию **ParallelFFT-Calculation.**

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size)
{
  FFTTask& a = *new(task::allocate_root()) FFTTask(signal, size);
  task::spawn_root_and_wait(a);
}
```

Отметим, что код функции **ParallelFFT** не изменится по сравнению с предыдущей реализацией.

Осталось внести необходимые изменения в функцию **main**.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  for (int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    ParallelFFT(inputSignal, outputSignal, size);
    duration  = (tick_count::now() - startTime).seconds();

    if (duration < minDuration)
      minDuration = duration;
  }

  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Задание 6. Проведение вычислительных экспериментов

Выполните сборку приложения в конфигурации **Release**. Проведите вычислительные эксперименты с тремя параллельными версиями:

- с простым подбором значения **grainsize** при использовании функции **parallel_for**;
- с адаптивным подбором значения **grainsize** при использовании функции **parallel_for**;
- с использованием логических задач.

Заполните таблицу 4. На основе значений, записанных в таблицу 4, посчитайте ускорения и заполните таблицу 5. Убедитесь в том, что наилучшие результаты показывает версия, реализованная на основе механизма логических задач библиотеки TBB.

Таблица 4

Результаты вычислительных экспериментов для вычисления БПФ

Номер теста	Размер входного сигнала	Время работы последовательного приложения (сек)	Время работы параллельного приложения 1 (сек)	Время работы параллельного приложения 2 (сек)	Время работы параллельного приложения 3 (сек)
1	32768				
2	65536				
3	131072				
4	262144				
5	524288				

Таблица 5

Ускорение вычислений, получаемое для параллельного вычисления БПФ

Номер теста	Размер входного сигнала	Ускорение параллельного алгоритма 1	Ускорение параллельного алгоритма 2	Ускорение параллельного алгоритма 3
1	32768			
2	65536			
3	131072			
4	262144			
5	524288			

8.6. Контрольные вопросы

1. Как собрать приложение, использующее библиотеку TBB?
2. Как инициализировать библиотеку TBB?
3. Как завершить работу библиотеки TBB?
4. Как инициализировать библиотеку TBB с определенным числом программных потоков?
5. Как оценить полученные результаты ускорения приложения?
6. В каких случаях нужно программировать с использованием логических задач?
7. Что такое функтор?
8. Каков смысл параметра grainsize?
9. Как работает конструкция parallel_for в библиотеке TBB?
10. Что такое логическая задача в библиотеке TBB?
11. В чем состоят особенности разработки параллельных версий алгоритмов с использованием логических задач?

8.7. Задания для самостоятельной работы

1. Реализуйте параллельную версию бит-реверсирования. Оцените вклад в ускорение, который внесет такая реализация.

2. Параллельная версия, разработанная с использованием логических задач, реализована таким образом, что последняя итерация вычислений выполняется последовательно. Реализуйте вычисление БПФ так, чтобы последняя итерация тоже выполнялась параллельно.

8.8. Литература

Использованные источники

1. Сиднев А.А., Сысоев А.В., Мееров И.Б. Библиотека Intel Threading Building Blocks – краткое описание // Материалы образовательного комплекса «Технологии разработки параллельных программ». Нижний Новгород, 2007.
2. Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
3. Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.
4. Гонсалес Р., Вудс Р. Цифровая обработка сигналов. – М.: Техносфера, 2005.

Рекомендуемая литература

5. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003 (Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. Reading, MA: Addison-Wesley, 2000).
6. Quinn M.J. Parallel Programming in C with MPI and OpenMP. New York, NY: McGraw-Hill, 2004.
7. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
8. Гергель В.П. Теория и практика параллельных вычислений. М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
9. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.

Дополнительная литература

10. Юнаковский А.Д. Начала вычислительных методов для физиков. Н. Новгород: ИПФ РАН, 2007.
11. Березин И.С., Жидков И.П. Методы вычислений. М.: Наука, 1966.
12. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. СПб.: Питер, 2006.
13. Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. М.: ДМК Пресс, 2006.
14. Павловская Т.А. C/C++. Программирование на языке высокого уровня. СПб.: Питер, 2003.
15. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows: Пер. с англ. 4-е изд. СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.

Информационные ресурсы сети Интернет

16. Страница библиотеки TBB на сайте корпорации Intel – [http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm].
17. Сайт сообщества пользователей TBB – [http://threadingbuildingblocks.org].
18. Сайт Лаборатории параллельных информационных технологий НИВЦ МГУ – [http://www.parallel.ru].
19. Официальный сайт OpenMP – [www.openmp.org].

8.9. Приложения

8.9.1. Программный код последовательного вычисления БПФ

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <complex>
#include <time.h>
#include "tbb/tick_count.h"

using namespace tbb;
using namespace std;

#define PI (3.14159265358979323846)

// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[ size - size/4 ]=1;
}

// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}

// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if(size < 4)
      cout << "Input signal length should be >= 4" << endl;
    else
    {
```

```
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two"
               << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  //Computing experiments
  //RandomDataInitialization(inputSignal, size);
  //-------------------------------------------------

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}

void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;

  delete [] outputSignal;
  outputSignal = NULL;
}

void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
```

```
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}

__inline void Butterfly(complex<double> *signal,
  complex<double> &u, int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}

void SerialFFTCalculation(complex<double> *signal, int size)
{
  int m = 0;
  for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++)
    //size = 2^m
    ;
  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    double coeff = PI / butterflySize;

    for (int i = 0; i < size / butterflyOffset; i++)
      for (int j = 0; j < butterflySize; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)),
          j + i * butterflyOffset, butterflySize);
  }
}

// FFT computation
void SerialFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  BitReversing(inputSignal, outputSignal, size);
  SerialFFTCalculation(outputSignal, size);
}

void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}

void PrintAmplitude(complex<double> *signal, int size)
{
```

```
  for(int i = 0; i < size; i++)
    cout << abs(signal[i]) << endl;
}

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  for(int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    // FFT computation
    SerialFFT(inputSignal, outputSignal, size);
    duration  = (tick_count::now() - startTime).seconds();

    if(duration < minDuration)
      minDuration = duration;
  }

  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

8.9.2. Программный код параллельного вычисления БПФ с использованием функции tbb::parallel_for

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <complex>
#include <time.h>
#include "tbb/tick_count.h"
#include "tbb/parallel_for.h"
```

```
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"

using namespace tbb;
using namespace std;

#define PI (3.14159265358979323846)

// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[size - size / 4] = 1;
}

// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}

// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if (size < 4)
      cout << "Input signal length should be >= 4" << endl;
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two"
               << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
  while (size < 4);
  cout << "Input signal length = " << size << endl;
```

```
  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  //Computing experiments
  //RandomDataInitialization(inputSignal, size);
  //-------------------------------------------------

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}

void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;

  delete [] outputSignal;
  outputSignal = NULL;
}

void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}

__inline void Butterfly(complex<double> *signal,
  complex<double> &u, int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
```

```
}

class ButterflyCalculator
{
  complex<double> *const signal;
  int const butterflySize;
  double coeff;

public:
   void operator()( const blocked_range<int>& r ) const
   {
     int begin = r.begin(),
         end = r.end(),
         butterflyOffset = 2 * butterflySize;

     for (int i = begin; i != end; i++)
       for (int j = 0; j < butterflySize; j++)
         Butterfly(signal, complex<double>(cos(-j * coeff),
           sin(-j * coeff)),
           j + i * butterflyOffset, butterflySize);
  }

  ButterflyCalculator(complex<double> *tsignal,
    int tbutterflySize):
    signal(tsignal), butterflySize(tbutterflySize)
  {
    coeff = PI / butterflySize;
  }
};

void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  int m = 0;
  for(int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++)
    // size = 2^m
    ;

  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    int grain = grainsize;

    // Adaptive grainsize
    // grain = grainsize/ butterflyOffset;
    // if (grain < 1)
    //  grain = 1;

    parallel_for(blocked_range<int>(0, size / butterflyOffset,
      grain), ButterflyCalculator(signal, butterflySize));
  }
}

void ParallelFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size, int grainsize)
```

```
{
  BitReversing(inputSignal, outputSignal, size);
  ParallelFFTCalculation(outputSignal, size, grainsize);
}

void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}

void PrintAmplitude(complex<double> *signal, int size)
{
  for(int i=0; i<size; i++)
    cout << abs(signal[i]) << endl;
}

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int testCount = 17;
  const int repeatCount = 16;
  tick_count startTime;
  double duration;

  cout << setprecision(6);
  for(int j = 0; j < testCount; j++)
  {
    int grainsize = size >> j;
    if (grainsize < 1)
      break;

    double minDuration = DBL_MAX;

    for(int i = 0; i < repeatCount; i++)
    {
      startTime = tick_count::now();
      ParallelFFT(inputSignal, outputSignal, size, grainsize);
      duration = (tick_count::now() - startTime).seconds();

      if (duration < minDuration)
        minDuration = duration;
    }
```

```
    cout << "Grain size is " << grainsize << ",
      execution time is ";
    cout << minDuration << " s. " << endl;
  }

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

8.9.3. Программный код параллельного вычисления БПФ с использованием логических задач

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <complex>
#include <time.h>
#include "tbb/tick_count.h"
#include "tbb/task.h"
#include "tbb/task_scheduler_init.h"

using namespace tbb;
using namespace std;

#define PI (3.14159265358979323846)

// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[size - size / 4] = 1;
}

// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}

// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
```

```
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if(size < 4)
      cout << "Input signal length should be >= 4" << endl;
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two"
               << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  //Computing experiments
  //RandomDataInitialization(inputSignal, size);
  //-------------------------------------------------

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}

void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;

  delete [] outputSignal;
  outputSignal = NULL;
}

void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
```

```
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}

__inline void Butterfly(complex<double> *signal,
  complex<double> &u, int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}

void SerialFFTCalculation(complex<double> *const signal, int size)
{
  if (size == 1)
    return;

  double const coeff = 2.0 * PI / size;

  SerialFFTCalculation(signal, size / 2);
  SerialFFTCalculation(&(signal[size / 2]), size / 2);

  for (int j = 0; j < size / 2; j++)
    Butterfly(signal, complex<double>(cos(-j * coeff),
      sin(-j * coeff)), j , size / 2);
}

class FFTTask: public task
{
  complex<double> *const signal;
  int const size;
  double coeff;

public:
  FFTTask(complex<double> *const tSignal, int tSize):
    signal(tSignal), size(tSize)
  {
    coeff = 2.0 * PI / size;
  }

  task* execute()
```

```
  {
    if (size < 1024)
      SerialFFTCalculation(signal, size);
    else
    {
      FFTTask& a =
        *new (allocate_child()) FFTTask(signal, size / 2);
      FFTTask& b =
        *new (allocate_child()) FFTTask(&(signal[size / 2]),
        size/2);

      set_ref_count(3);

      spawn(b);
      spawn_and_wait_for_all(a);

      for (int j = 0; j < size / 2; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j , size / 2);
    }

    return NULL;
  }
};

void ParallelFFTCalculation(complex<double> *signal, int size)
{
  FFTTask& a = *new(task::allocate_root()) FFTTask(signal, size);
  task::spawn_root_and_wait(a);
}

void ParallelFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  BitReversing(inputSignal, outputSignal, size);
  ParallelFFTCalculation(outputSignal, size);
}

void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}

void PrintAmplitude(complex<double> *signal, int size)
{
  for(int i = 0; i < size; i++)
    cout << abs(signal[i]) << endl;
}

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;
```

```
  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data initialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  for (int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    ParallelFFT(inputSignal, outputSignal, size);
    duration = (tick_count::now() - startTime).seconds();

    if (duration < minDuration)
      minDuration = duration;
  }

  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Корняков Кирилл Владимирович,
Мееров Иосиф Борисович,
Сиднев Алексей Александрович,
Сысоев Александр Владимирович,
Шишков Александр Валерьевич

ИНСТРУМЕНТЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ

Учебное пособие

Под ред. проф. В.П. Гергеля

Бумага офсетная. Печать офсетная. Формат 70×108 1/16.
Уч.-изд. л. 19,3. Усл. печ. л. 17,7. Тираж 300 экз. Заказ № 344.

Оригинал-макет подготовлен отделом дизайна
редакционно-издательского управления (РИУ)
Нижегородского госуниверситета им. Н.И. Лобачевского.
603950, Н. Новгород, пр. Гагарина, 23.

Издательство Нижегородского госуниверситета им. Н.И. Лобачевского.
603950, Н. Новгород, пр. Гагарина, 23.

Отпечатано в типографии Нижегородского госуниверситета.
603000, Н. Новгород, ул. Б. Покровская, 37.