Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

А.Н. Калитаев
Л.Г. Егорова
В.Е. Торчинский

МЕТОДЫ И СРЕДСТВА ВЫСОКОПРОИЗВОДИТЕЛЬНОГО ПРОГРАММИРОВАНИЯ

*Утверждено Редакционно-издательским советом университета
в качестве практикума*

Магнитогорск
2022

УДК 004.43

Рецензенты:

технический директор
ООО «Центр интернет-технологий «Факт»
Е.А. Гарбар

доктор технических наук, доцент,
заведующий кафедрой автоматизированных систем управления
ФГБОУ ВО «Магнитогорский государственный
технический университет им. Г.И. Носова»
С.М. Андреев

Калитаев А.Н., Егорова Л.Г., Торчинский В.Е.
Методы и средства высокопроизводительного программирования [Электронный ресурс]: практикум /Александр Николаевич Калитаев, Людмила Геннадьевна Егорова, Вадим Ефимович Торчинский : ФГБОУ ВО «Магнитогорский государственный технический университет им. Г.И. Носова». – Электрон. текстовые дан. (0,81 Мб). – Магнитогорск : ФГБОУ ВО «МГТУ им. Г.И. Носова», 2022. – 1 электрон. опт. диск (CD–R). – Систем. требования : IBMPC, любой, более lGHz; 512 Мб RAM ; 10 Мб HDD ; MSWindowsXP и выше ; AdobeReader8.0 и выше; CD/DVD–ROM дисковод ; мышь. – Загл. с титул. экрана.

Данное учебное издание представляет собой практикум по дисциплине «Методы и средства высокопроизводительного программирования» для магистрантов направления 09.04.01 – «Информатика и вычислительная техника».

В практикуме рассматриваются: классификация, архитектура современных вычислительных систем и методики оценки их производительности, подход в организации параллельных вычислений, реализуемых средствами Windows API; технология OpenMP, как одна из наиболее популярных средств программирования для компьютеров с общей памятью. Описание функциональности WinAPI и OpenMP при организации высокопроизводительных вычислений в данном практикуме сопровождается примерами и лабораторными работами для самостоятельного изучения и выполнения студентами на практике.

УДК 004.43

Содержание

ВВЕДЕНИЕ

В настоящее время круг задач, требующих для своего решения применения мощных вычислительных ресурсов, все время расширяется. Высокопроизводительные вычисления не мыслятся без распараллеливания, ибо наиболее мощные вычислительные системы имеют сотни и тысячи процессоров, работающих одновременно и в тесном взаимодействии, т.е. параллельно. При организации проведения высокопроизводительных вычислений стало обычным использование многозадачности и мультипрограммности, мультимедийных средств, компьютерных локальных сетей, а также глобальных сетей, таких, как Интернет. Это показывает, что серьезное изучение вопросов распараллеливания и высокопроизводительных вычислений чрезвычайно важно. Немаловажным аспектом в высокопроизводительном программировании является знание классификации, архитектуры современных вычислительных систем и методик оценки их производительности.

В практикуме рассматривается подход в организации параллельных вычислений, реализуемых средствами Windows API (Application Programming Interface). В контексте исполнения процесса могут выполняться несколько потоков, а поток — это единица исполнения, которой ОС выделяет процессорное время для выполнения программы. В операционной системе Windows существуют средства, позволяющие использовать системные ресурсы непосредственно в прикладных программах. Эти средства объединены в совокупность системных процедур и функций, принадлежащих ядру ОС и ее надстройкам.

Также в практикуме рассматривается технология OpenMP, как одна из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. При этом создаваемая параллельная программа будет переносимой между различными компьютерами с разделяемой памятью, поддерживающими OpenMP API.

Описание функциональности WinAPI и OpenMP при организации высокопроизводительных вычислений в данном практикуме сопровождается примерами и лабораторными работами для самостоятельного изучения и выполнения студентами на практике.

ЛАБОРАТОРНАЯ РАБОТА №1.
СРАВНЕНИЕ КОМПИЛЯТОРОВ СОВРЕМЕННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ И ИХ ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ ПК

Цель работы

Изучить классификацию многопроцессорных вычислительных систем и оценить влияние компиляторов современных языков программирования на производительность вычислительных систем.

Информация

Одним из наиболее распространенных способов классификации ЭВМ является систематика Флинна, в рамках которой основное внимание при анализе архитектуры вычислительных систем уделяется способам взаимодействия последовательностей (потоков) выполняемых команд и обрабатываемых данных [1-5]. Схематично классификация Флинна показана на рисунке 1.

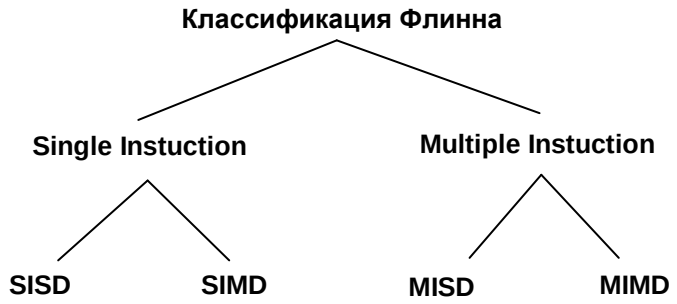


Рис. 1. Классификация вычислительных систем

SISD (Single Instruction, Single Data) – системы, в которых существует одиночный поток команд и одиночный поток данных. К данному типу систем можно отнести обычные последовательные компьютеры.

SIMD (Single Instruction, Multiple Data) – системы с одиночным потоком команд и множественным потоком данных. SIMD компьютеры состоят из одного командного процессора (управляющего модуля), называемого контроллером, и нескольких модулей обработки данных, называемых процессорными элементами.

MISD (Multiple Instruction, Single Data) – системы, в которых существует множественный поток команд и одиночный поток данных. Вычислительных машин такого класса практически нет и трудно привести пример их успешной реализации.

MIMD (Multiple Instruction, Multiple Data) – системы с множественным потоком команд и множественным потоком данных. К подобному классу систем относится большинство параллельных многопроцессорных высокопроизводительных вычислительных систем.

В категории MIMD проводят классификацию вычислительных систем (рисунок 2):

–мультипроцессоры (один компьютер со многими процессорами);

–мультикомпьютеры (вычисления проводятся на относительно независимых компьютерах).

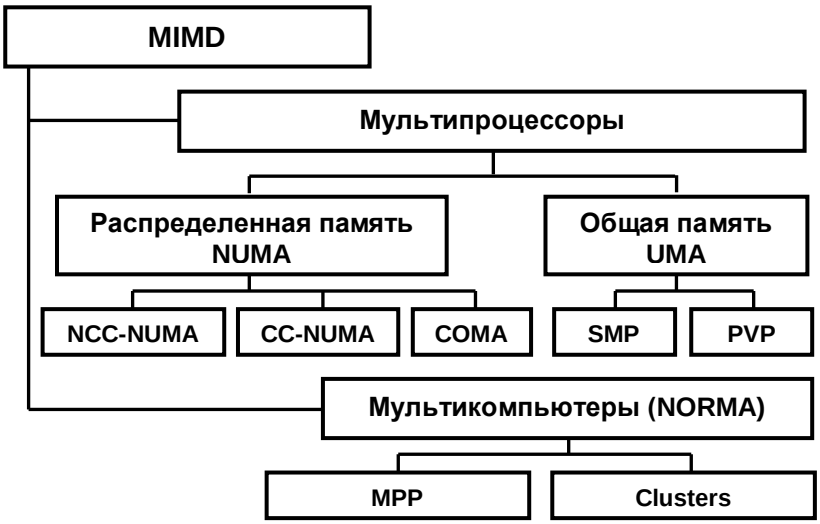


Рис. 2. Классификация MIMD систем

При классификации мультипроцессоров учитывается способ построения общей памяти. Возможный подход – использование единой (централизованной) общей памяти. Такой подход обеспечивает однородный доступ к памяти (uniform memory access - UMA) и служит основой для построения векторных параллельных процессоров (parallel vector processor - PVP) и симметричных мультипроцессоров (symmetric multiprocessor - SMP).

Общий доступ к данным может быть обеспечен и при физически распределенной памяти. При этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти. Такой подход именуется как неоднородный доступ к памяти (non-uniform memory access - NUMA). Среди систем с таким типом памяти выделяют три типа систем. Системы, в которых для представления данных используется только локальная кэш-память имеющихся процессоров (cache-only memory architecture - COMA). Системы, в которых обеспечивается когерентность локальной кэш-памяти разных процессоров (cache-coherent NUMA - CC-NUMA). И, наконец, системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэш-памяти (non-cache coherent NUMA - NCC-NUMA). Использование распределенной общей памяти (distributed shared memory - DSM) упрощает проблемы создания мультипроцессоров. Известны примеры систем с несколькими тысячами процессоров.

Мультикомпьютеры - многопроцессорные системы с распределенной памятью - уже не обеспечивают общий доступ ко всей имеющейся в системах памяти (no-remote memory access - NORMA). При всей схожести подобной

архитектуры с системами с распределенной общей памятью, мультикомпьютеры имеют принципиальное отличие – каждый процессор системы может использовать только свою локальную память. При этом для доступа к данным, располагаемых на других процессорах, необходимо явно выполнить операции передачи сообщений (message passing operations). Данный подход используется при построении двух важных типов многопроцессорных вычислительных систем - массивно-параллельных систем (massively parallel processor - MPP) и кластеров (clusters). Это системы с распределенной памятью и с произвольной коммуникационной системой.

Задание

Решить задачи на 3 трех языках программирования, оценить качество сделанных программ и провести сравнительную характеристику.

Условие задач:

1. Составить алгоритм и программу для нахождения количества четных и нечетных N-значных чисел, состоящих из цифр, которые попарно являются соседними в натуральном ряду. N задается пользователем (например, N=8, N=10).

2. Составить алгоритм и программу для нахождения всех «совершенных» чисел в диапазоне целых чисел от 1 до N (например, N=1000000). «Совершенным» называется число, равное, сумме всех своих делителей, исключая само число. Например: 28 = 1+2+4+7+14.

Выполнение работы:

Программы составить на языках программирования C++, C#, Visual Basic в среде программирования Microsoft Visual Studio.

Задание: Оценить качество программ и занести результаты оценки в таблицу 1:

Таблица 1.

Компьютер	Язык программи-рования	Количество найденных чисел	Время выполнения*, сек	
			с учетом оптими-зации	без учета оптими-зации
n-значность числа *n*=				
	C++			
	C#			
	Visual Basic			
	C++			
	C#			
	Visual Basic			
n-значность числа *n*=				
	C++			
	C#			
	Visual Basic			
	C++			
	C#			
	Visual Basic			

* - учитывается среднее время выполнения программы по 3 экспериментам, в случае, если время не может быть определено (например, 0 мс), определяем суммарное время за m – экспериментов (например, m = 100).

ЛАБОРАТОРНАЯ РАБОТА №2.
ПРЯМОЕ СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПК

Цель работы

Изучить методики оценки производительности вычислительных систем и оценить производительность вычислительных систем с использованием реализованных тестов на вычисления с вещественными и целыми числами

Информация

Основу для сравнения различных типов компьютеров между собой дают стандартные методики измерения производительности. В процессе развития вычислительной техники появилось несколько стандартных методик. Они позволяют разработчикам и пользователям осуществлять выбор между альтернативами на основе количественных показателей, что дает возможность постоянного прогресса в данной области [5].

Единицей измерения производительности компьютера является время: компьютер, выполняющий тот же объем работы за меньшее время является более быстрым. Время выполнения любой программы измеряется в секундах. Часто производительность измеряется как скорость появления некоторого числа событий в секунду, так что меньшее время подразумевает большую производительность.

Однако в зависимости от того, что мы считаем, время может быть определено различными способами. Наиболее простой способ определения времени называется астрономическим временем, временем ответа (response time), временем выполнения(execution time) или прошедшим временем (elapsed time). Это задержка выполнения задания, включающая буквально все: работу процессора, обращения к диску, обращения к памяти, ввод/вывод и накладные расходы операционной системы. Однако при работе в мультипрограммном режиме во время ожидания ввода/вывода для одной программы, процессор может выполнять другую программу, и система не обязательно будет минимизировать время выполнения данной конкретной программы.

Для измерения времени работы процессора на данной программе используется специальный параметр - время центрального процессора (CPU time), которое не включает время ожидания ввода/вывода или время выполнения другой программы. Очевидно, что время ответа, видимое пользователем, является полным временем выполнения программы, а не временем центрального процессора (ЦП). Время ЦП может далее делиться на время, потраченное ЦП непосредственно на выполнение программы пользователя и называемое пользовательским временем ЦП, и время ЦП, затраченное операционной системой на выполнение заданий, затребованных программой, и называемое системным временем ЦП.

В ряде случаев системное время ЦП игнорируется из-за возможной неточности измерений, выполняемых самой операционной системой, а также из-за проблем, связанных со сравнением производительности машин с разными операционными системами. С другой стороны, системный код на некоторых машинах является пользовательским кодом на других и, кроме того,

практически никакая программа не может работать без некоторой операционной системы. Поэтому при измерениях производительности процессора часто используется сумма пользовательского и системного времени ЦП.

В большинстве современных процессоров скорость протекания процессов взаимодействия внутренних функциональных устройств определяется не естественными задержками в этих устройствах, а задается единой системой синхросигналов, вырабатываемых некоторым генератором тактовых импульсов, как правило, работающим с постоянной скоростью. Дискретные временные события называются тактами синхронизации (clock ticks), просто тактами (ticks), периодами синхронизации (clock periods), циклами (cycles) или циклами синхронизации (clock cycles).Разработчики компьютеров обычно говорят о периоде синхронизации, который определяется либо своей длительностью (например, 10 наносекунд), либо частотой (например, 100 МГц). Длительность периода синхронизации есть величина, обратная к частоте синхронизации.

Таким образом, время ЦП для некоторой программы может быть выражено двумя способами: количеством тактов синхронизации для данной программы, умноженным на длительность такта синхронизации, либо количеством тактов синхронизации для данной программы, деленным на частоту синхронизации.

Важной характеристикой, часто публикуемой в отчетах по процессорам, является среднее количество тактов синхронизации на одну команду - CPI (clock cycles per instruction). При известном количестве выполняемых команд в программе этот параметр позволяет быстро оценить время ЦП для данной программы.

Таким образом, производительность ЦП зависит от трех параметров: такта (или частоты) синхронизации, среднего количества тактов на команду и количества выполняемых команд. Невозможно изменить ни один из указанных параметров изолированно от другого, поскольку базовые технологии, используемые для изменения каждого из этих параметров, взаимосвязаны: частота синхронизации определяется технологией аппаратных средств и функциональной организацией процессора; среднее количество тактов на команду зависит от функциональной организации и архитектуры системы команд; а количество выполняемых в программе команд определяется архитектурой системы команд и технологией компиляторов. Когда сравниваются две машины, необходимо рассматривать все три компоненты, чтобы понять относительную производительность.

В процессе поиска стандартной единицы измерения производительности компьютеров было принято несколько популярных единиц измерения: MIPS - (миллион команд в секунду), MFLOPS (миллионах чисел-результатов вычислений с плавающей точкой в секунду, или миллионах элементарных арифметических операций над числами с плавающей точкой, выполненных в секунду), но в действительности единственной подходящей и надежной

единицей измерения производительности является время выполнения реальных программ.

Задание

Для сравнения производительности ПК реализовать тесты на вычисления с вещественными (т.е. дробными) числами и вычисления с целыми числами. Для представления вещественных чисел можно использовать тип double (8 байт), а для целых – DWORD (unsigned long) (4 байта).

1. Тесты с вычислениями над числами с плавающей точкой.

1.1. Вычисления числа π с помощью ряда:

$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \pi/4.$

Компьютер	Время выполнения, с

1.2. Вычисление интеграла функции $\sin(x)$ при $0 \leq x \leq \pi$ методом прямоугольников.

Компьютер	Время выполнения, с

2. Тесты с вычислениями над целыми числами. При этом, тесты обрабатывающие массивы данных, необходимо проводить для разных размеров массивов, так как скорость выполнения таких задач зависит не только от вычислительной мощности процессора, но в не меньшей мере и от размера и скорости кэша, скорости памяти, эффективности реализации всей подсистемы памяти в целом.

2.1. Возведение в квадрат единичной матрицы. Чтобы точнее измерить время для небольших матриц, тест выполняет расчеты по несколько раз. Необходимо рассмотреть 2 размерности матрицы, например, 10000x10000 (а); 100x100 (б).

Компьютер	Время (а), с	Время (б), с

2.2. Сортировка псевдослучайной последовательности чисел пузырьковым методом. Необходимо провести операцию для последовательностей разной длины: 50000 (а) элементов; 500 (б). Чтобы точнее измерить время для небольших последовательностей, тест выполняет расчеты по несколько раз.

Компьютер	Время (а), с	Время (б), с

2.3. Вычисление суммы всех простых чисел, меньших, чем N, где N, в данном случае равно 50000. Особенность этой задачи состоит в том, что все данные умещаются в регистрах процессора. Таким образом, сравнивается исключительно производительность ядер.

Компьютер	Время выполнения, с

Для сравнения производительности ПК необходимо привести таблицу с «нормированными» значениями. То есть значения буду рассчитывать по формуле $v = vt \cdot h/1000$, где vt - время, полученное в тесте, h - частота процессора. Таким образом, такое значение показывала бы данная система на частоте 1 ГГц, при условии линейной зависимости производительности от частоты.

Компьютер	Время выполнения, с						
	1.1	1.2	2.1а	2.1б	2.2а	2.2б	2.3

ЛАБОРАТОРНАЯ РАБОТА №3.
ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ WINAPI

Цель работы

Получение практических навыков по использованию Win32/64 API для организации параллельных вычислений.

Информация

В операционных системах (ОС) семейства Windows NT (32- и 64-разрядных) существуют средства, позволяющие использовать системные ресурсы вычислительной техники для организации параллельных вычислений. Эти средства объединены в совокупность системных процедур и функций, принадлежащих ядру ОС и ее надстройкам [1, 6-7]. Множество этих процедур и функций получило название интерфейса прикладного программирования API.

1. Принципы работы с потоками в ОС Windows

Процесс пользовательского режима ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потока выполняется в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах [2].

Поток (*thread*) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Большинство приложений обходится единственным, первичным потоком. Однако процессы могут создавать дополнительные потоки, что позволяет им эффективнее выполнять свою работу.

Каждый поток начинает выполнение с входной функции. В первичном потоке таковым является main, wmain, WinMain или wWinMain. Если необходимо создать вторичный поток, в нем должна быть входная функция, которая выглядит следующим образом:

```
DWORD WINAPI ThreadFunc(PVOID lpParam){
DWORD  dwResult=0;
...
return dwResult;
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент поток остановится, память, отведенная под его стек, будет освобождена, а счетчик пользователей

объекта ядра «поток» уменьшится на *1*. Когда счетчик обнулится, этот объект ядра будет разрушен.

Создание потока

Для создания дополнительных потоков необходимо вызвать из первичного потока функцию CreateThread:

HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId);

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке.

Параметр lpThreadAttributes является указателем на структуру SECURITY_ATTRIBUTES. Если необходимо, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), в этом параметре передается NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, необходимо определить структуру SECURITY_ATTRIBUTES и инициализировать ее элемент bInheritHandle значением TRUE.

Параметр dwStackSize определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Если при обращении к функции CreateThread, передается в параметре dwStackSize ненулевое значение, функция резервирует всю указанную память. Ее объем определяется либо значением параметра dwStackSize, либо значением, заданным в ключе /STACK (/STACK:[reserve][,commit] – аргумент reserve определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию – 1Мб); аргумент commit задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию – 1 страница)) компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в dwStackSize. Если же в параметре dwStackSize передается нулевое значение, CreateThread создает стек для нового потока, используя информацию, встроенную компоновщиком в EXE-файл.

Параметр lpStartAddress определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр lpParameter идентичен параметру lpParameter функции потока. CreateThread лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать

функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией. Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же функции. Например, можно реализовать Web-сервер, который обрабатывает каждый клиентский запрос в отдельном потоке. При создании каждому потоку передается свое значение lpParameter. Так как Windows – операционная система с вытесняющей многозадачностью, следовательно, новый поток и поток, вызвавший CreateThread, могут выполняться одновременно, что может привести к определенным проблемам.

Параметр dwCreationFlags определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код.

Последний параметр lpThreadId функции CreateThread – это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку.

Завершение потока

Поток можно завершить четырьмя способами [2]:

– функция потока возвращает управление (рекомендуемый способ);

– поток самоуничтожается вызовом функции *ExitThread* (нежелательный способ);

VOID ExitThread(DWORD dwExitCode);

В параметре dwExitCode помещается значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, так как после ее вызова поток перестает существовать.

– один из потоков данного или стороннего процесса вызывает функцию TerminateThread для завершения потока с дескриптором hThread (нежелательный способ);

BOOL TerminateThread(HANDLE hThread,
DWORD dwExitCode);

– завершается процесс (с использованием функций: ExitProcess и TerminateProcess), содержащий данный поток (нежелательный способ).

Дополнительные функции для работы с потоками:

– SuspendThread – функция приостановки работы потока с дескриптором hThread:

DWORD SuspendThread(HANDLE hThread);

– ResumeThread – функция возобновления работы потока с дескриптором hThread:

DWORD ResumeThread(HANDLE hThread);

– Sleep – функция приостановки работы потока на определенный период времени (dwMilliseconds – количество миллисекунд):

VOID Sleep(DWORD dwMilliseconds);

– GetThreadPriority – функция позволяет узнать относительный приоритет потока с дескриптором hThread:

int GetThreadPriority(HANDLE hThread);

– SetThreadPriority – функция изменения относительного приоритета потока с дескриптором hThread:

BOOL SetThreadPriority(HANDLE hThread,
int nPriority);

параметр nPriority передается один из идентификаторов, соответствующий определенному приоритету потока:

THREAD_PRIORITY_TIME_CRITICAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_IDLE

2. Механизмы синхронизации операционной системы Windows

В Win32 существуют средства синхронизации двух типов:

– реализованные на уровне пользователя (критические секции – Critical section);

– реализованные на уровне ядра (мьютексы – Mutex, события – Event, семафоры – Semaphore).

Общие черты механизмов синхронизации:

– используют примитивы ядра при выполнении, что сказывается на производительности;

– могут быть именованными и неименованными;

– работают на уровне системы, то есть могут служить механизмом межпроцессного взаимодействия;

– используют для ожидания и захвата примитива единую функцию: WaitForSingleObject/WaitForMultipleObjects.

Существуют несколько стратегий, которые могут применяться, чтобы разрешать проблемы, связанные с взаимодействием потоков.

Наиболее распространенным способом является синхронизация потоков, суть которой состоит в том, чтобы вынудить один поток ждать, пока другой не закончит какую-то определенную заранее операцию. Для этой цели существуют специальные синхронизирующие объекты ядра операционной системы *Windows*. Они исключают возможность одновременного доступа к тем данным, которые с ними связаны. Их реализация зависит от конкретной ситуации и предпочтений программиста.

Общие положения использования объектов ядра системы:

– однажды созданный объект ядра можно открыть в любом приложении, если оно имеет соответствующие права доступа к нему;

– каждый объект ядра имеет счетчик числа своих пользователей. Как только он станет равным нулю, система уничтожит объект ядра;

– обращаться к объекту ядра надо через описатель (handle), который система дает при создании объекта;

– каждый объект может находиться в одном из двух состояний: свободном (signaled) или занятом (nonsignaled).

Работа с объектом Критическая секция (Critical section)

Критическая секция (*critical section*) – это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент вытеснить поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый ресурс, не получит процессорное время до тех пор, пока поток не выйдет за границы критической секции. Критические секции являются простыми объектами ядра Windows, которые не снижают общей эффективности приложения.

Для работы с критическими секциями есть ряд функций API и тип данных CRITICAL_SECTION. Алгоритм использования следующий:

1. Объявить глобальную структуру CRITICAL_SECTION cs.

2. Инициализировать (обычно это делается один раз, перед тем как начнется работа с разделяемым ресурсом) глобальную структуру вызовом функции InitializeCriticalSection (&cs).

3. Поместить охраняемую часть программы внутрь блока, который начинается вызовом функции EnterCriticalSection и заканчивается вызовом LeaveCriticalSection:

```
EnterCriticalSection (&cs);
{
  // охраняемый блок кодов
}
LeaveCriticalSection (&cs);
```

Функция EnterCriticalSection, анализируя поле структуры cs, которое является счетчиком ссылок, выясняет, вызвана ли она в первый раз.

Если да, то функция увеличивает значение счетчика и разрешает выполнение потока дальше. При этом выполняется блок, модифицирующий критические данные. Допустим, в это время истекает квант времени, отпущенный данному потоку, или он вытесняется более приоритетным потоком, использующим те же данные.

Новый поток выполняется, пока не встречает функцию EnterCriticalSection, которая помнит, что объект cs уже занят. Новый поток приостанавливается (засыпает), а остаток процессорного времени передается другому потоку.

Функция LeaveCriticalSection уменьшает счетчик ссылок на объект *cs*.

Как только поток покидает критическую секцию, счетчик ссылок обнуляется и система будит ожидающий поток, снимая защиту секции кодов.

Критические секции применяются для синхронизации потоков лишь в пределах одного процесса. Они управляют доступом к данным так, что в каждый конкретный момент времени только один поток может их изменять.

4. Когда надобность в синхронизации потоков отпадает, следует вызвать функцию, освобождающую все ресурсы, включенные в критическую секцию: DeleteCriticalSection (&cs).

Таким образом, поток, который желает обезопасить определенные данные от race conditions, вызывает функцию EnterCriticalSection / TryEnterCriticalSection:

– если критическая секция свободна, поток занимает ее;

– если же нет, поток блокируется до тех пор, пока секция не будет освобождена другим потоком с помощью вызова функции LeaveCriticalSection.

Данные функции – атомарные, то есть целостность данных нарушена не будет.

Пример. Разграничение доступа к общему ресурсу с использованием критической секции

```
int g_nNums[100]; // разделяемый ресурс
CRITICAL_SECTION g_cs; // защита ресурса
DWORD WINAPI ThreadFunc(PVOID pvParam) {
  EnterCriticalSection(&g_cs);
  for (int x = 0; x < 100; x++)
  {
    g_nNums[x] = 0;
  }
  LeaveCriticalSection(&g_cs);
  return 0;
}
```

Работа с объектом Семафор (Semaphore)

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Для семафоров определены следующие правила:

– когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;

– если этот счетчик равен 0, семафор занят;

– система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;

– счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Для работы с объектом Semaphore существует ряд функций:

– функция CreateSemaphore() создает семафор с заданным начальным значением счетчика и максимальным значением, которое ограничивает доступ;

HANDLE CreateSemaphore(PSECURITY_ATTRIBUTE psa,
LONG lInitialCount,
LONG lMaximumCount,
PCTSTR pszName);

Параметр psa является указателем на структуру SECURITY_ATTRIBUTES, в большинстве случаев параметр принимает значение NULL, при этом создается объект с защитой по умолчанию. Параметр lMaximumCount сообщает системе максимальное число ресурсов, обрабатываемое приложением. Поскольку 32-битное значение со знаком, предельное число ресурсов может достигать 2147483647. Параметр lInitialCount указывает, сколько из этих ресурсов доступно изначально (на данный момент). Параметр pszName определяет имя объекта ядра Windows (если параметр принимает значение NULL, то создается безымянный объект ядра).

– функция OpenSemaphore() осуществляет доступ к семафору;

HANDLE OpenSemaphore(DWORD fdwAccess,
BOOL bInheritHandle,
PCTSTR pszName);

– функция ReleaseSemaphore() увеличивает значение счетчика. Счетчик может меняться от 0 до максимального значения;

BOOL ReleaseSemaphore(HANDLE hSem,
LONG lReleaseCount,
PLONG plPreviousCount);

– после завершения работы надо вызвать CloseHandle().

BOOL CloseHandle(HANDLE hSem);

Работа с объектом Мьютекс (Mutex)

Объекты ядра «мьютексы» (*mutual exclusion, mutex*) гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока. Мьютексы ведут себя точно так же, как и критические секции. Однако, если последние являются объектами пользовательского режима, то мьютексы – объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Для работы с этим объектом есть ряд функций:

– функция создания объекта Mutex – CreateMutex();

HANDLE CreateMutex(PSECURITY_ATTRIBUTES psa,
BOOL fInitialOwner,
PCTSTR pszName);

Параметр fInitialOwner определяет начальное состояние мьютекса. Если в нем передается FALSE, объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1.

Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

– для доступа – OpenMutex();

HANDLE OpenMutex(DWORD fdwAccess,
BOOL bInheritHandle,
PCTSTR pszName);

для освобождения ресурса – ReleaseMutex()*;*

BOOL ReleaseMutex(HANDLE hMutex);

Функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать ReleaseMutex столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится.

– для доступа к объекту Mutex используется ожидающая функция WaitForSingleObject().

Каждая программа создает объект Mutex по имени, то есть Mutex – это именованный объект.

С любым объектом ядра сопоставляется счетчик, фиксирующий, сколько раз данный объект передавался во владение потокам.

Объект Mutex отличается от других синхронизирующих объектов ядра тем, что занявшему его потоку передаются права на владение им.

Прочие синхронизирующие объекты могут быть либо свободны, либо заняты и только, а Mutex способны еще и запоминать, какому потоку они принадлежат.

Отказ от Mutex происходит, когда ожидавший его поток захватывает этот объект, переводя его в занятое состояние, а потом завершается. В таком случае получается, что Mutex занят и никогда не освободится, поскольку другой поток не сможет этого сделать. Система не допускает подобных ситуаций и, заметив, что произошло, автоматически переводит Mutex в свободное состояние.

Работа с объектом Событие (Event)

События – самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая – его состояние (свободен или занят). События просто уведомляют об окончании какой-либо операции.

Для работы с объектом Event есть ряд функций:

– функция CreateEvent() используется для создания события;

HANDLE CreateEvent(PSECURITY_ATTRIBUTES psa,
BOOL fManualReset,
BOOL fInitialState,
PCTSTR pszName);

Параметр fManualReset (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE).

Параметр fInitialState определяет начальное состояние события – свободное (TRUE) или занятое (FALSE).

– функция OpenEvent() – для доступа;

HANDLE OpenEvent(DWORD fdwAccess,
BOOL fInherit,
PCTSTR pszName);

– функция SetEvent() – для установки события в свободное состояние;

BOOL SetEvent(HANDLE hEvent);

– функция ResetEvent() используется для сброса события.

BOOL ResetEvent(HANDLE hEvent);

– функция PulseEvent() освобождает событие и тут же переводит его обратно в занятое состояние; ее вызов равнозначен последовательному вызову SetEvent() и ResetEvent().

BOOL PulseEvent(HANDLE hEvent);

Дескриптор события после окончания работы нужно закрыть. Класс CEvent представляет функциональность синхронизирующего объект ядра (события). Он позволяет одному потоку уведомить (*notify*) другой поток о том, что произошло событие, которое тот поток, возможно, ждал.

Существуют два типа объектов: ручной (*manual*) и автоматический (*automatic*):

– ручной объект начинает сигнализировать, когда будет вызван метод SetEvent. Вызов ResetEvent переводит его в противоположное состояние;

– автоматический объект класса CEvent не нуждается в сбросе. Он сам переходит в состояние nonsignaled, и охраняемый код при этом недоступен, когда хотя бы один поток был уведомлен о наступлении события.

Пример. Использование объектов ядра «событие» для синхронизации потоков.

```
// глобальный описатель события со сбросом вручную
//(в занятом состоянии)
HANDLE g_hEvent;
int WINAPI WinMain(...) {
// создаем объект «событие со сбросом вручную»
//(в занятом состоянии)
g_hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
// порождаем три новых потока
HANDLE hThread[3];
DWORD dwThreadID;
hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadID);
hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadID);
hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0,
&dwThreadID);
OpenFileAndReadContentsIntoMemory(...);
// разрешаем всем трем потокам обращаться к памяти
```

```
SetEvent(g_hEvent);
}
DWORD WINAPI WordCount(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
return 0;
}
DWORD WINAPI SpellCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
return 0;
}
DWORD WINAPI GrammarCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
return 0;
}
```

При запуске этот процесс создает занятое событие со сбросом вручную и записывает его описатель в глобальную переменную. Это упрощает другим потокам процесса доступ к тому же объекту-событию. Затем порождается три потока. Они ждут, когда в память будут загружены данные (текст) из некоего файла, и потом обращаются к этим данным: например, один поток подсчитывает количество слов, другой проверяет орфографические ошибки, третий – грамматические. Все три функции потоков начинают работать одинаково: каждый поток вызывает WaitForSingleObject(), которая приостанавливает его до тех пор, пока первичный поток не считает в память содержимое файла.

Функции ряда Wait

Функции ряда Wait блокируют выполнение потока до наступления какого-то события или тайм-аута. Для того чтобы пользоваться этими функциями, должен быть объект синхронизации, который проверяет эти функции. У этих объектов два состояния: «установлен» и «сброшен».

Алгоритм использования ожидающих функций:

– вызов функций (им передается указатель на объект синхронизации);

– объект проверяется;

– если объект не установлен, то функция будет ждать, пока не истечет тайм-аут, все это время поток будет блокирован.

Сценарий синхронизации потоков с использованием ожидающих функций:

– прежде чем заснуть, поток сообщает системе то особое событие, которое должно разбудить его;

– при засыпании потока операционная система перестает выделять ему кванты процессорного времени, приостанавливая его выполнение;

– как только указанное потоком событие произойдет, система возобновит выдачу ему квантов процессорного времени и поток вновь может развиваться.

Win32 API поддерживает целый ряд функций, которые начинаются с Wait:

– *WaitForMultipleObjects*;

– *WaitForMultipleObjectsEx*;

– *WaitForSingleObject*;

– *WaitForSingleObjectEx*;

– *MsgWaitForMultipleObjects*;

– *MsgWaitForMultipleObjectsEx*.

Также существует функция WaitCommEvent(), предназначенная для работы с данными в последовательных портах.

Функции, у которых в имени есть Single, предназначены для установки одного синхронизирующего объекта.

Функции, у которых в имени есть Multiple, используются для установки ожидания сразу нескольким объектам.

Функции с префиксами Msg предназначены для ожидания события определенного типа, например, ввода с клавиатуры.

Функции с окончанием Ex расширены опциями по управлению асинхронным вводом-выводом.

Использование функций ряда Wait для синхронизации потоков

Потоки «усыпляют» себя до освобождения какого-либо синхронизирующего объекта с помощью следующих функций ряда Wait:

DWORD WaitForSingleObject (HANDLE hObject,
DWORD dwTimeOut)

DWORD WaitForMultipleObjects (DWORD nCount,
CONST HANDLE lpHandles,*
BOOL bWaitAll,
DWORD dwTimeOut)

Функция WaitForSingleObject приостанавливает поток до тех пор, пока:

– заданный параметром hObject синхронизирующий объект не освободится;

– не истечет интервал времени, задаваемый параметром dwTimeOut. Если указанный объект в течение заданного интервала не перейдет в свободное состояние, то система вновь активизирует поток, и он продолжит свое выполнение.

В качестве параметра dwTimeOut могут выступать два особых значения:

– 0 – функция только проверяет состояние объекта (занят или свободен) и сразу же возвращается;

– INFINITE – время ожидания бесконечно; если объект так и не освободится, поток останется в неактивном состоянии и никогда не получит процессорного времени.

Функция WaitForSingleObject в соответствии с причинами, по которым поток продолжает выполнение, может возвращать одно из следующих значений:

– WAIT_TIMEOUT – объект не перешел в свободное состояние, но интервал времени истек;

– WAIT_ABANDONED – ожидаемый объект является Mutex, который не был освобожден владеющим им потоком перед окончанием этого потока. Объект Mutex автоматически переводится системой в состояние свободен. Такая ситуация называется «отказ от Mutex»;

– WAIT_OBJECT_0 – объект перешел в свободное состояние;

– WAIT_FAILED – произошла ошибка, причину которой можно узнать, вызвав GetLastError.

Пример. Работа функции WaitForSingleObject со значением таймаута, отличным от INFINITE:

```
DWORD dw = WaitForSingleObject(hThread, 5000);
switch (dw) {
  case WAIT_OBJECT_0: // поток завершается
    break;
  case WAIT_TIMEOUT:
    // поток не завершился в течение 5000 мс
    break;
  case WAIT_FAILED:
   //неправильный вызов функции (неверный описатель?)
    break;
}
```

Функция WaitForMultipleObjects задерживает поток и, в зависимости от значения флага bWaitAll, ждет одного из следующих событий:

– освобождение хотя бы одного синхронизирующего объекта из заданного списка;

– освобождение всех указанных объектов;

– истечение заданного интервала времени.

Пример. Работа функции WaitForMultipleObjects с тремя потоками:

```
HANDLE h[3];
h[0] = hThread1;
h[1] = hThread2;
h[2] = hThread3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw) {
  case WAIT_FAILED:
   //неправильный вызов функции (неверный описатель?)
    break;
  case WAIT_TIMEOUT:
```

```
//ни один из объектов не освободился в течение 5000 мс
    break;
  case WAIT_OBJECT_0 + 0:
    // завершился поток, идентифицируемый h[0],
    //т.е. описателем (hThread1)
    break;
  case WAIT_OBJECT_0 + 1:
    // завершился поток, идентифицируемый h[1],
    //т.е. описателем (hThread2)
    break;
  case WAIT_OBJECT_0 + 2:
    // завершился поток, идентифицируемый h[2],
    //т.е. описателем (hThread3)
  break;}
```

Часть ОС, называемая системным планировщиком (*system scheduler*), управляет переключением заданий, определяя, какому из конкурирующих потоков следует выделить следующий квант времени процессора. Решение принимается с учетом приоритетов конкурирующих потоков. Множество приоритетов, определенных в операционной системе для потоков, занимает диапазон от 0 (низший приоритет) до 31 (высший приоритет).

При наличии нескольких процессоров Windows применяет симметричную модель распределения потоков по процессорам symmetric multiprocessing (*SMP*). Это означает, что любой поток может быть направлен на любой процессор. Программист может ввести некоторые коррективы в эту модель равноправного распределения. Функции SetProcessAffinityMask и SetThreadAffinityMask позволяют указать предпочтения в смысле выбора процессора для всех потоков процесса или для одного определенного потока. Потоковое предпочтение (*thread affinity*) вынуждает систему выбирать процессоры только из множества, указанного в маске.

Существует также возможность для каждого потока указать один предпочтительный процессор. Это делается с помощью функции SetThreadidealProcessor. Это указание служит подсказкой для планировщика заданий.

Пример. Пример параллельных вычислений числа ПИ по формуле Грегори с использованием Windows API (распараллеливание на gNumThreads = *2* потоков с синхронизацией на примере использования критических секций).

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

const long long gNumSteps = (long long)2e9;
const unsigned short gNumThreads = 2;
long double gStep = 0.0, gPi = 0.0;
CRITICAL_SECTION gCS;
```

```
DWORD WINAPI threadFunction(LPVOID pArg)
{
  unsigned short myNum = *((unsigned short *)pArg);
  long double partialSum = 0.0; // локальная переменная каждого //потока
  for (long long i=myNum; i< gNumSteps ; i+=gNumThreads)
// вычисление части суммы каждым потоком
  {
    partialSum += 1.0/(i*4.0 + 1.0);
    partialSum -= 1.0/(i*4.0 + 3.0);
  }
// добавление части суммы к суммарной глобальной //переменной
(итоговому значению числа ПИ)
  EnterCriticalSection(&gCS);
  gPi += 4.0*partialSum;
  LeaveCriticalSection(&gCS);
  return 0;
}

int main()
{
  HANDLE threadHandles[gNumThreads];
  unsigned short tNum[gNumThreads];
  DWORD time_begin = GetTickCount();
  printf("\n Computed value of Pi: ");
  InitializeCriticalSection(&gCS);
  gStep = 1.0 / gNumSteps;
  for (unsigned short i=0; i < gNumThreads; i++)
  {
    tNum[i] = i;
    threadHandles[i]=CreateThread(NULL,0,// Размер стека
      threadFunction,  // Функция потока
      (LPVOID)&tNum[i],// Данные, передаваемые в функцию потока
      0, NULL);       // Возвращаемое значение ID потока
  }
  WaitForMultipleObjects(gNumThreads,threadHandles, TRUE,INFINITE);
  DeleteCriticalSection(&gCS);
  printf("%12.10lf\n", gPi );
  printf_s(" time = %.3lf sec.",(double)(GetTickCount()-
time_begin)/CLOCKS_PER_SEC);
  getchar();
  return 0;
}
```

Результаты работы программы представлены на рис. 3а, 4а, 5а, а на рис. 3б, 4б, 5б представлена информация по загрузке центрального и логических процессоров во время выполнения расчетов.

а)

```
Computed value of Pi: 3.1415926533
time = 46.238 sec.
```

б)

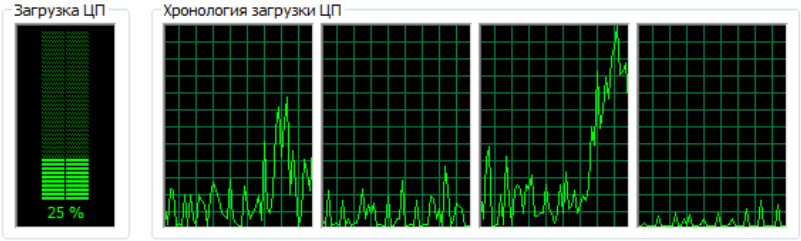


Рис. 3 Результат работы программы (а) и информация по загрузке ЦП (б) при вычислении значения числа π без распараллеливания ($gNumThreads = 1$)

а)

```
Computed value of Pi: 3.1415926533
time = 23.869 sec.
```

б)

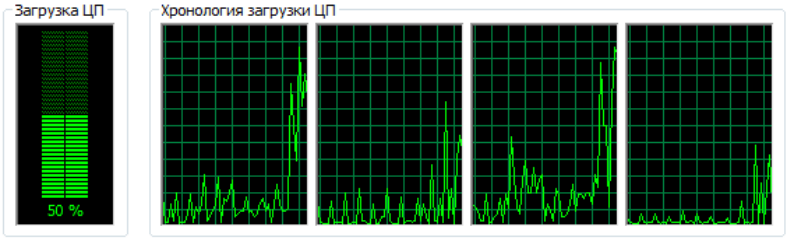


Рис. 4. Результат работы программы (а) и информация по загрузке ЦП (б) при вычислении значения числа π при распараллеливании на 2 потока ($gNumThreads = 2$)

а)

```
Computed value of Pi: 3.1415926533
time = 13.135 sec.
```

б)

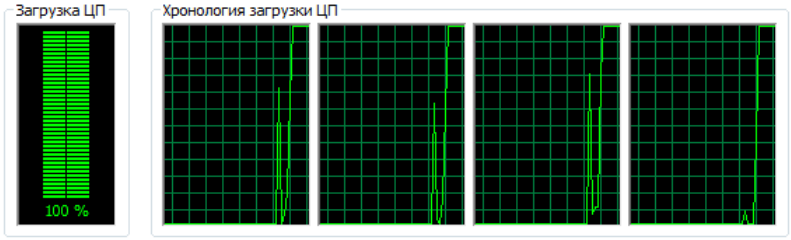


Рис. 5. Результат работы программы (а) и информация по загрузке ЦП (б) при вычислении значения числа π при распараллеливании на 4 потока ($gNumThreads = 4$)

Задание

Написать многопоточную программу для вычисления приближенного значения интеграла $\int_a^b f(x)dx$ методами прямоугольников, трапеций и парабол (варианты заданий представлены в табл. 2 [3]).

Программа запускает n потоков, которые начинают вычислять значение определенного интеграла по заданной формуле, согласно, методу численного интегрирования, главный поток ожидает завершения вычислений всеми потоками, выводит значение и время вычислений на экран. Потоки, работая

параллельно, используют объекты синхронизации для разграничения доступа и изменения значения общей переменной предназначенной для хранения результата вычислений.

Результаты вычислений при разном количестве потоков привести в табл. 3 и выполнить сравнительный анализ результатов.

Таблица 2

Варианты заданий

№ вари-анта	Интеграл	№ вари-анта	Интеграл
1.	$f(x)=\int_1^{3,5}\frac{\ln(x)}{x\sqrt{1+\ln(x)}}dx$	2.	$f(x)=\int_2^3\frac{1}{x\cdot ln(x)}dx$
3.	$f(x)=\int_{\pi/6}^{\pi/3}(tg^2(x)+ctg^2(x))dx$	4.	$f(x)=\int_1^4\frac{ln^2(x)}{x}dx$
5.	$f(x)=\int_1^{\ln(2)}\sqrt{e^x-1}dx$	6.	$f(x)=\int_1^4\frac{dx}{\sqrt{9+x^2}}$
7.	$f(x)=\int_0^1 x\cdot e^x\cdot\sin(x)dx$	8.	$f(x)=\int_0^{\sqrt{3}} x\cdot arctg(x)dx$
9.	$f(x)=\int_0^1\frac{\lvert x^2-1\rvert}{(x^2+1)\sqrt{x^4+1}}dx$	10.	$f(x)=\int_1^e(x\cdot ln(x))^2dx$
11.	$f(x)=\int_0^{1,5}\frac{e^x\cdot(1+\sin(x))}{1+\cos(x)}dx$	12.	$f(x)=\int_0^3 arcsin\sqrt{\frac{x}{1+x}}dx$

Таблица 3

Сравнительный анализ результатов

Метод вычисления значения интеграла функции	Значение и время вычисления интеграла функции $f(x)$ $(a\le x\le b)$				
	значе-ние	время расчета при k-потоках, с			
		$k=1$	$k=2$	$k=4$	$k=8$
Метод прямоугольников					
Метод трапеций					
Метод парабол (Симпсона)					

ЛАБОРАТОРНАЯ РАБОТА №4.
ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ OPENMP

Цель работы

Изучение и приобретение практических навыков параллельного программирования с использованием технологии OpenMP, базирующейся на использовании концепции параллельных данных для компьютеров с общей памятью.

Информация

Одним из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Предполагается, что создаваемая параллельная программа будет переносимой между различными компьютерами с разделяемой памятью, поддерживающими OpenMP API [5,8].

Технология OpenMP нацелена на то, чтобы пользователь имел один вариант программы для параллельного и последовательного выполнения. Однако возможно создавать программы, которые работают корректно только в параллельном режиме или дают в последовательном режиме другой результат. Более того, из-за накопления ошибок округления результат вычислений с использованием различного количества нитей может в некоторых случаях различаться.

Разработкой стандарта занимается некоммерческая организация OpenMP ARB (Architecture Review Board) [1], в которую вошли представители крупнейших компаний – разработчиков SMP-архитектур и программного обеспечения. OpenMP поддерживает работу с языками Fortran и C/C++. Первая спецификация для языка Fortran появилась в октябре 1997 года, а спецификация для языка C/C++ – в октябре 1998 года. На данный момент последняя официальная спецификация стандарта – OpenMP 5.2 [12] (принята в ноябре 2021 года).

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA и других) в модели общей памяти (shared memory model). На рис. 6 представлена схема многопроцессорных систем с общей памятью. В стандарт OpenMP входят спецификации набора директив компилятора, вспомогательных функций и переменных среды. OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор «подчиненных» (slave) потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами, причём количество процессоров не обязательно должно быть больше или равно количеству потоков.

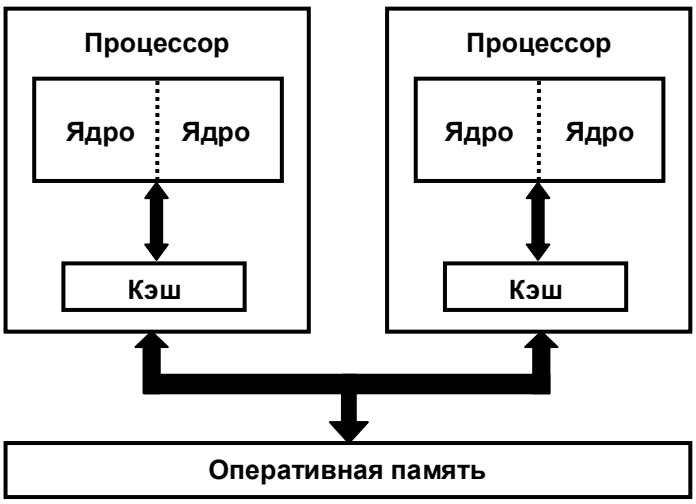


Рис. 6. Схема многопроцессорных систем с общей памятью

POSIX-интерфейс для организации нитей (Pthreads) поддерживается практически на всех UNIX-системах, однако по многим причинам не подходит для практического параллельного программирования: в нём нет поддержки языка Фортран, слишком низкий уровень программирования, нет поддержки параллелизма по данным, а сам механизм нитей изначально разрабатывался не для целей организации параллелизма. OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей); в OpenMP используется терминология и модель программирования, близкая к Pthreads, например, динамически порождаемые нити, общие и разделяемые данные, механизм «замков» для синхронизации.

Согласно терминологии POSIX threads, любой UNIX-процесс состоит из нескольких нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. В простейшем случае процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).

Важным достоинством технологии OpenMP является возможность реализации так называемого инкрементального программирования, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, в программе нераспараллеленная часть постепенно становится всё меньше. Такой подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию.

Положительные стороны технологии OpenMP

1. Поэтапное (инкрементальное) распараллеливание. Можно распараллеливать последовательные программы поэтапно, не меняя их структуру.

2. Единственность разрабатываемого кода. Нет необходимости поддерживать одновременно последовательный и параллельный вариант

программы, поскольку директивы игнорируются обычными компиляторами (в общем случае).

3. Эффективность. Учет и использование возможностей систем с общей памятью.

4. Переносимость. Поддержка большим числом компиляторов под разные платформы и ОС, стандарт для распространенных языков C/C++.

Принципы организации параллелизма

1. Использование потоков (общее адресное пространство).

2. Пульсирующий (fork-join) параллелизм.

3. При выполнении обычного кода (вне параллельных областей) программа исполняется одним потоком (master thread).

4. При появлении директивы #parallel происходит создание “команды” (team) потоков для параллельного выполнения вычислений.

5. После выхода из области действия директивы #parallel происходит синхронизация, все потоки, кроме master, уничтожаются.

6. Продолжается последовательное выполнение кода (до очередного появления директивы #parallel).

На рисунке 7 представлена многопоточность, когда главный поток разделяет несколько потоков, которые выполняют блоки кода параллельно.

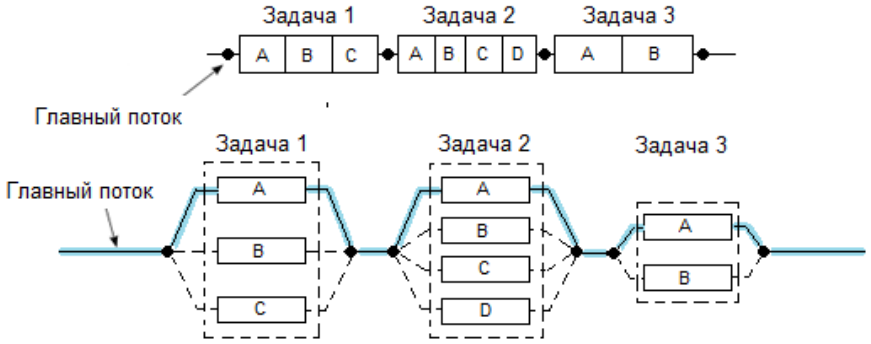


Рис. 7. Многопоточность при параллельном выполнении кода

Структура

1. Набор директив компилятора.

2. Библиотека функций.

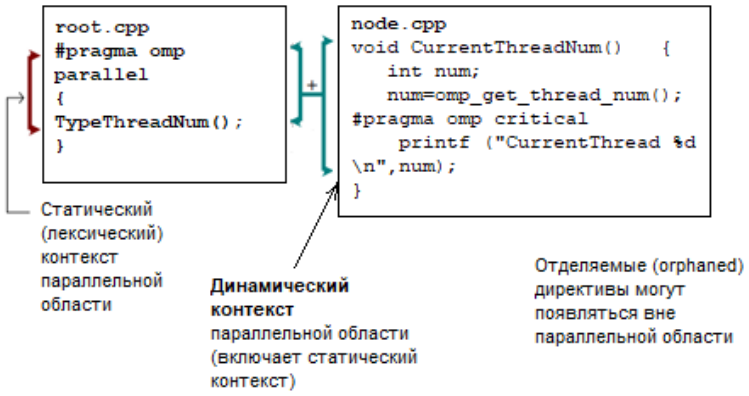
3. Набор переменных окружения.

Директивы OpenMP

Формат записи директив на примере C/C++.

#pragma omp имя_директивы [clause,…]

#pragma omp parallel default(shared) private(beta,pi)



Типы директив

1. Определение параллельной области
2. Разделение работы
3. Синхронизация

Определение параллельной области

1. Директива parallel (основная директива OpenMP).

2. Когда основной поток выполнения достигает директиву parallel, создается набор (team) потоков; входной поток является основным потоком этого набора (master thread) и имеет номер 0.

3. Код области дублируется или разделяется между потоками для параллельного выполнения.

4. В конце области обеспечивается синхронизация потоков – выполняется ожидание завершения вычислений всех потоков; далее все потоки завершаются – дальнейшие вычисления продолжает выполнять только основной поток.

5. Формат директивы parallel

 # pragma omp parallel [clause …] newline structured_block

Возможные параметры (clause)

If (scalar_expression)
private (list)
firstprivate (list)
default (shared | none)
shared (list)
copyin (list)
reduction (operator: list)
num_threads (integer-expression)

1. Количество потоков (по убыванию старшинства)

– num_threads(N).
– omp_set_num_threads().
– OMP_NUM_THREADS.
– Число, равное количеству процессоров, которое “видит” операционная система.

2. Параметр (clause) if – если условие в if не выполняется, то процессы не создаются.

Пример использования директивы parallel

```
int main ()
{
int nthreads, tid;
// Создание параллельной области
#pragma omp parallel private(tid)
{
// печать номера потока
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
// Печать количества потоков – только master
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} // Завершение параллельной области
return 0;
}
```

Управление областью видимости данных

1. Управление областью видимости обеспечивается при помощи параметров (clause) директив:

– *private,*
– *firstprivate,*
– *lastprivate,*
– *shared,*
– *default,*
– *reduction,*
– *copyin,*

которые определяют, какие соотношения существуют между переменными последовательных и параллельных фрагментов выполняемой программы

2. Параметр shared определяет список переменных, которые будут общими для всех потоков параллельной области; правильность использования таких переменных должна обеспечиваться программистом

#pragma omp parallel shared(list).

3. Параметр private определяет список переменных, которые будут локальными для каждого потока; переменные создаются в момент формирования потоков параллельной области; начальное значение переменных является неопределенным

#pragma omp parallel private(list).

Пример использования директивы parallel

```
int main ()
{
int nthreads, tid;
// Создание параллельной области
#pragma omp parallel private(tid)
```

```
{
// печать номера потока
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
// Печать количества потоков – только master
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} // Завершение параллельной области
return 0;
}
```

4. Параметр firstprivate позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных

#pragma omp parallel firstprivate(list).

5. Параметр lastprivate позволяет создать локальные переменные потоков, значения которых запоминаются в исходных переменных после завершения параллельной области (используются значения потока, выполнившего последнюю итерацию цикла или последнюю секцию)

#pragma omp parallel lastprivate(list).

Распределение вычислений между потоками

1. Существует три директивы для распределения вычислений в параллельной области:

– DO / for – распараллеливание циклов

– sections – распараллеливание раздельных фрагментов кода (функциональное распараллеливание)

– single – директива для указания последовательного выполнения кода

2. Начало выполнения директив по умолчанию не синхронизируется.

3. Завершение директив по умолчанию является синхронным.

4. Формат директивы for

#pragma omp for [clause ...] newline
for loop

Возможные параметры (clause)

private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered schedule(kind[, chunk_size])
nowait

Распределение итераций в директиве for регулируется параметром (clause) schedule**.**

– static – итерации делятся на блоки по chunk итераций и статически разделяются между потоками; если параметр chunk не определен, итерации делятся между потоками равномерно и непрерывно;

– dynamic – распределение итерационных блоков осуществляется динамически (по умолчанию chunk=1);

– guided – размер итерационного блока уменьшается экспоненциально при каждом распределении; chunk определяет минимальный размер блока (по умолчанию chunk=1);

– runtime – правило распределения определяется переменной OMP_SCHEDULE (при использовании runtime параметр chunk задаваться не должен).

Пример использования директивы for

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
int main ()
{
int i, n, chunk;
float a[NMAX], b[NMAX], c[NMAX];
for (i=0; i < NMAX; i++){
a[i] = i * 1.0;
b[i] = a[i];
  }
n = NMAX;
chunk = CHUNK;
#pragma omp parallel shared(a,b,c,n,chunk) private(i)
{
#pragma omp for schedule(dynamic,chunk) nowait
for (i=0; i < n; i++)
c[i] = a[i] + b[i];
} // end of parallel section
return 0;
}
```

5. Формат директивы sections

```
#pragma omp sections [clause ...] newline
{
#pragma omp section newline
structured_block
#pragma omp section newline
structured_block
}
```

Возможные параметры (clause)

```
private(list)
firstprivate(list)
lastprivate(list)
```

reduction(operator: list

nowait

Директива sections – распределение вычислений для раздельных фрагментов кода:

– фрагменты выделяются при помощи директивы section;

– каждый фрагмент выполняется однократно;

– разные фрагменты выполняются разными потоками;

– завершение директивы по умолчанию синхронизируется;

– директивы section должны использоваться только в статическом контексте.

Пример использования директивы sections

```
#include<omp.h>
#define NMAX 1000
int main ()
{
int i, n;
float a[NMAX], b[NMAX], c[NMAX];
for (i=0; i < NMAX; i++){
  a[i] = i * 1.0;
  b[i] = a[i];
}
n = NMAX;
#pragma omp parallel shared(a,b,c,n) private(i)
{
#pragma omp sections nowait
{
#pragma omp section
for (i=0; i < n/2; i++)
c[i] = a[i] + b[i];
#pragma omp section
for (i=n/2; i < n; i++)
c[i] = a[i] + b[i];
} // end of sections
} // end of parallel section
return 0;
}
```

6. Объединение директив parallel и for/sections

```
#include<omp.h>
#define CHUNK 100
#define NMAX 1000
int main ()
{
int i, n, chunk;
float a[NMAX], b[NMAX], c[NMAX];
for (i=0; i < NMAX; i++){
```

```
a[i] = i * 1.0;
b[i] = a[i];
}
n = NMAX;
chunk = CHUNK;
#pragma omp parallel for shared(a,b,c,n)  schedule(static,chunk)
for (i=0; i < n; i++)
c[i] = a[i] + b[i];
  return 0;
}
```

Операция редукции

Параметр reduction определяет список переменных, для которых выполняется операция редукции:

1. Перед выполнением параллельной области для каждого потока создаются копии этих переменных;
2. Потоки формируют значения в своих локальных переменных;
3. При завершении параллельной области на всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных.

reduction (operator: list)

Пример использования параметра reduction

```
#include<omp.h>
int main ()
{
// vector dot product
int i, n, chunk;
float a[100], b[100], result;
n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++)
{
a[i] = i * 1.0;
b[i] = i * 2.0;
}
#pragma omp parallel for default(shared) schedule(static,chunk) reduction(+:result)
for (i=0; i < n; i++)
result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
return 0;
}
```

Правила записи параметра reduction

Возможный формат записи выражения:

x = x op expr

x = expr op x

x binop = expr

x++, ++x, x--, --x

–x должна быть скалярной переменной;

–expr не должно ссылаться на x;

–op (operator) должна быть неперегруженной операцией вида +, -, *, /, &, ^, |, &&, || ;

–binop должна быть неперегруженной операцией вида +, -, *, /, &, ^, |.

Синхронизация

Директива master определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется).

#pragma omp master newline

structured_block

Директива single определяет фрагмент кода, который должен быть выполнен только одним потоком (любым).

#pragma omp single [clause ...] newline

structured_block

Формат директивы single

#pragma omp single [clause ...] newline

structured_block

Возможные параметры (clause)

private(list)

firstprivate(list)

copyprivate(list)

nowait

Один поток исполняет блок в single, остальные потоки приостанавливаются до завершения выполнения блока.

Директива critical определяет фрагмент кода, который должен выполняться только одним потоком в каждый текущий момент времени (критическая секция).

#pragma omp critical [name] newline

structured_block

Пример использования директивы critical

```
#include<omp.h>
int main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
{
#pragma omp critical
x = x + 1;
} // end of parallel section
return 0;
```

}

Директива barrier – определяет точку синхронизации, которую должны достигнуть все процессы для продолжения вычислений (директива должны быть вложена в блок).

#pragma omp barrier newline.

Директива atomic – определяет переменную, доступ к которой (чтение/запись) должна быть выполнена как неделимая операция.

#pragma omp atomic newline

statement_expression

Возможный формат записи выражения x binop = expr , x++, ++x, x--, --x:

1. x должна быть скалярной переменной;
2. expr не должно ссылаться на x;
3. binop должна быть неперегруженной операцией вида +, -, *, /, &, ^, |, >>, <<.

Директива flush – определяет точку синхронизации, в которой системой должно быть обеспечено единое для всех процессов состояние памяти (т.е. если потоком какое-либо значение извлекалось из памяти для модификации, измененное значение обязательно должно быть записано в общую память).

#pragma omp flush (list) newline

Если указан список list, то восстанавливаются только указанные переменные.

Директива flush неявным образом присутствует в директивах barrier, critical, ordered, parallel, for, sections, single.

В таблице 4 приведена совместимость директив и их параметров.

Таблица 4

Совместимость директив и их параметров

Clause	Directive					
	PARALLEL	DO/ for	SEC-TIONS	SIN-GLE	PARAL-LEL DO/for	PARALLEL SECTIONS
IF	+				+	+
PRIVATE	+	+	+	+	+	+
SHARED	+	+			+	+
DEFAULT	+				+	+
FIRSTPRIVATE	+	+	+	+	+	+
LASTPRIVATE		+	+		+	+
REDUCTION	+	+	+		+	+
COPYIN	+				+	+
SCHEDULE		+			+	
ORDERED		+			+	
NOWAIT		+	+	+		

Этапы создания консольного приложения в среде MS Visual Studio с поддержкой OpenMP

Для включения поддержки OpenMP установите дополнительные параметры компиляции проекта:

1. В главном меню выберите Project-> Имя_проекта Properties

2. В открывшемся окне выберите Configuration Properties / C / C++ / Language. Установите для опции OpenMP Support значение Yes(/openmp). Нажмите кнопку OK (рисунок 8).

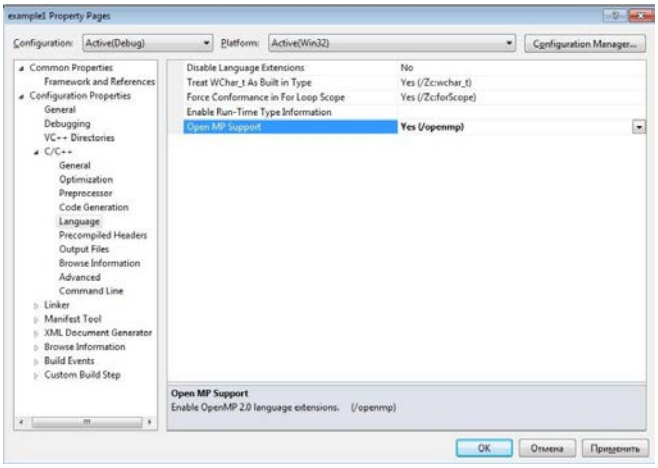


Рис. 8.Окно настройки параметров компиляции проекта
в MS Visual Studio

Тестовое задание. Напишите программу, в которой создается 4 потока и каждый поток выводит на экран строку "Hello World!".

Указания к выполнению задания:

1. Создайте консольное приложение в среде Visual Studio с поддержкой OpenMP.

2. Напишите на языке C/C++ программу, печатающую на экран строку «Hello World!».

3. Подключите заголовочный файл omp.h с функциями и переменными OpenMP.

Строка подключения заголовочного файла:

```
#include <omp.h>
```

4. В функции main создайте параллельную область с помощью OpenMP-директивы parallel. Обратите внимание, что открывающаяся фигурная скобка и название директивы должны находиться в разных строках! Поместите команду вывода строки «Hello World!» внутрь параллельной области.

```
#pragma omp parallel
{
  printf("Hello World!\n");
}
```

5. Задайте количество нитей в параллельной области одним из следующих способов:

Способ 1. Вызовите функцию omp_set_num_threads() перед началом параллельной области. В качестве параметра укажите одно целое число – количество нитей в параллельной области:

```
omp_set_num_threads(4);
#pragma omp parallel
{
  printf("Hello World!\n");
}
```

Способ 2. Добавьте к директиве parallel параметр num_threads(). В качестве параметра укажите одно целое число – количество нитей в параллельной области:

```
#pragma omp parallel num_threads(4)
{
  printf("Hello World!\n");
}
```

6. Скомпилируйте и запустите ваше приложение. Убедитесь, что строка «Hello World!» выводится на экран столько раз, сколько нитей вы задали в параллельной области.

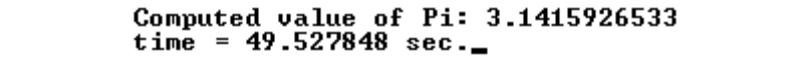
Пример. Вычисление числа ПИ по формуле Грегори с использованием OpenMP (распараллеливание на numThreads = 4 потока).

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

int main()
{
long long numSteps = 2e9, step;
long double valPi = 0.0, tmbegin, tmend;
unsigned short numThreads = 4;
tmbegin = omp_get_wtime();
#pragma omp parallel for private(step) reduction(+:valPi) num_threads(numThreads)
for(step = 0; step < numSteps; step++)
{
valPi += 1.0/(step * 4.0 + 1.0);
valPi -= 1.0/(step * 4.0 + 3.0);
}
valPi *= 4;
tmend =  omp_get_wtime();
printf("\n Computed value of Pi: %.10lf\n time = %lf sec.", valPi, tmend - tmbegin);
getchar();
return 0;
}
```

Результаты работы программы представлены на рис. 9а, 10а, 11а, а на рис. 9б, 10б, 11б представлена информация по загрузке центрального и логических процессоров во время выполнения расчетов.

а)



б)

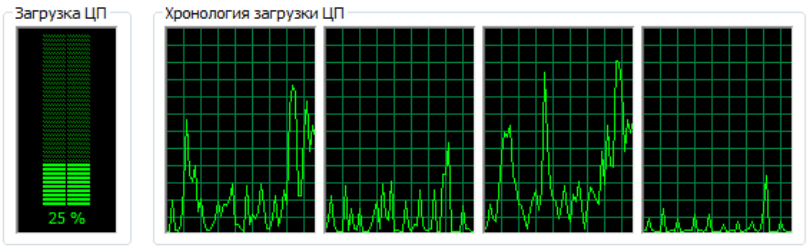
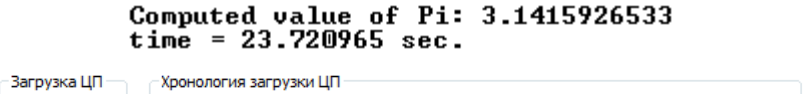


Рис. 9. Результат работы программы (а) и информация по загрузке ЦП (б) при вычислении значения числа π без распараллеливания ($numThreads = 1$)

а)



б)

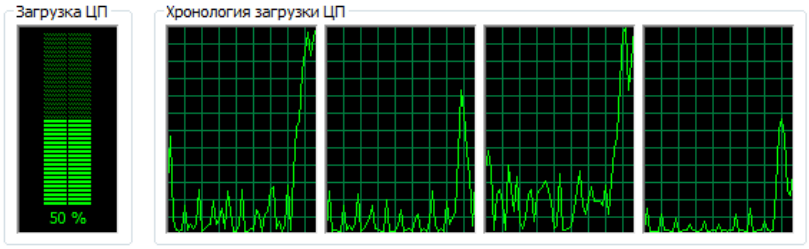
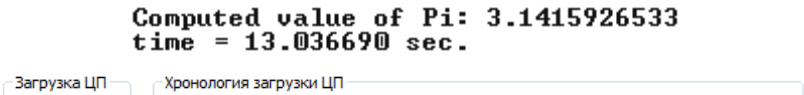


Рис. 10. Результат работы программы (а) и информация по загрузке ЦП (б) при вычислении значения числа π при распараллеливании на 2 потока ($numThreads = 2$)

а)



б)

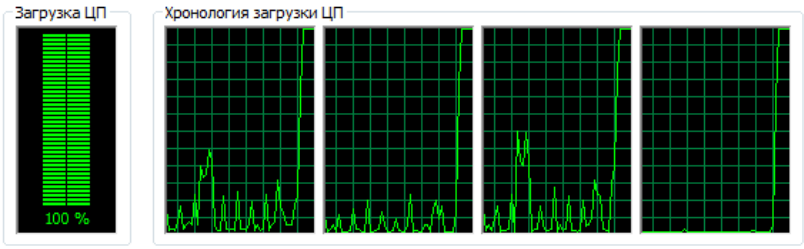


Рис. 11. Результат работы программы (а) и информация по загрузке ЦП (б) при вычислении значения числа π при распараллеливании на 4 потока ($numThreads = 4$)

Задание

Написать многопоточную программу для вычисления приближенного значения интеграла $\int_a^b f(x)dx$ методами прямоугольников, трапеций и парабол

(варианты заданий представлены в таблице 2 лабораторной работы №3) с использованием технологии OpenMP. Выполнить сравнительный анализ результатов работы с результатами, полученными в лабораторной работе №3.

Таблица 5

Сравнительный анализ результатов работы программы

Численный метод интегрирования	Значение и время вычисления интеграла функции $f(x)$ $(a \leq x \leq b)$						
	средствами WinAPI		средствами OpenMP				
	значение	время, с	значение	время расчета при k-потоках, с			
				$k = 1$	$k = 2$	$k = 4$	$k = 8$
Метод прямоугольников							
Метод трапеций							
Метод парабол							

ЛАБОРАТОРНАЯ РАБОТА №5.
OPENMP. МАТРИЧНЫЕ ВЫЧИСЛЕНИЯ И СОРТИРОВКА ДАННЫХ

Цель работы

Изучение и приобретение практических навыков параллельного программирования с использованием технологии OpenMP при организации матричных вычислений и сортировки данных.

Информация

Параллельные методы матричного умножения

Операция умножения матриц является одной из основных задач матричных вычислений [9-10]. Умножение матрицы A размером $m \cdot n$ и матрицы B размером $n \cdot l$ приводит к получению матрицы C размером $m \cdot l$, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l.$$

Каждый элемент результирующей матрицы C есть скалярное произведение соответствующей строки матрицы A и столбца матрицы B:

$$c_{ij} = \left(a_i, b_j^T\right), a_i = (a_{i0}, a_{i1}, \dots, a_{in-1}), b_j^T = \left(b_{0j}, b_{1j}, \dots, b_{n-1j}\right)^T.$$

Этот алгоритм предполагает выполнение $m \cdot n \cdot l$ операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера $n \cdot n$ количество выполненных операций имеет порядок $O(n^3)$.

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
// Последовательный алгоритм умножения матриц
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы C. Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной i) вычисляется одна строка результирующей матрицы (рис. 12).

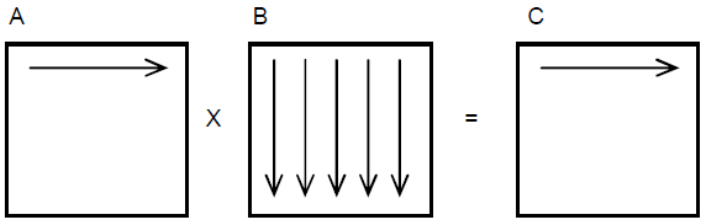
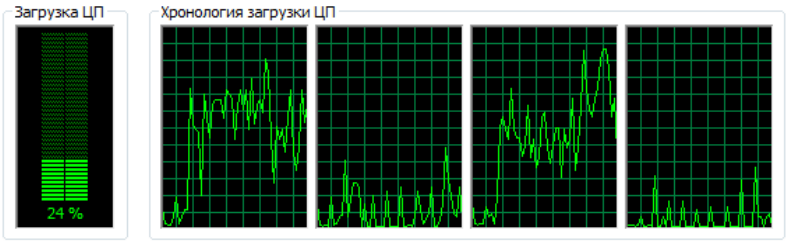


Рис. 12. Схема алгоритма вычисления элементов первой строки результирующей матрицы C при перемножении матриц A и B

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы C размером $n \cdot n$ необходимо выполнить $n^2 \cdot (n^2 - 1)$ скалярных операций.

```
// Параллельный алгоритм умножения матриц
#include <stdio.h>
#include <omp.h>
#define N 1000
double a[N][N], b[N][N], c[N][N];
int main()
{
  int i, j, k;
  double t1, t2;
// инициализация матриц
  for (i=0; i<N; i++)
    for (j=0; j<N; j++){
      a[i][j]=i*j;
      b[i][j]=a[i][j];
    }
  t1=omp_get_wtime();
// основной вычислительный блок
  #pragma omp parallel for shared(a, b, c) private(i, j, k)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      c[i][j] = 0.0;
      for(k=0; k<N; k++)
        c[i][j]+=a[i][k]*b[k][j];
    }
  }
  t2=omp_get_wtime();
  printf("Time=%lf\n", t2-t1);
  return 0;
}
```

а)



б)

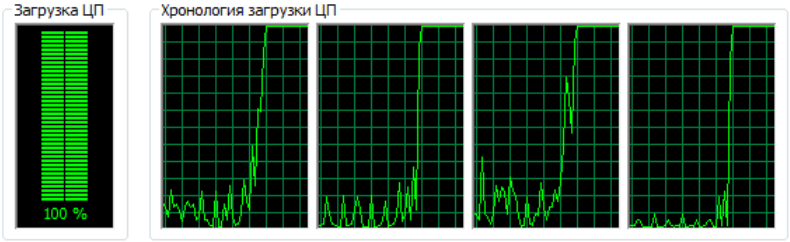


Рис. 13. Информация по загрузке ЦП при выполнении последовательного (а) и параллельного (б) (распараллеливание на 4 потока) алгоритмов умножения матриц

Сравнение времени выполнения последовательного и параллельного (распараллеливание на 4 потока) алгоритмов умножения матриц представлено в таблице 6. Эксперименты проводились на вычислительном узле на базе четырехядерного процессора Intel(R) Core(TM) i5-3210M CPU @ 2.50ГГц, 4Гб RAM.

Таблица 6

Сравнительный анализ результатов работы программы

Размер матриц	Время выполнения, сек.	
	последовательный алгоритм	параллельный алгоритм (4 потока)
1000x1000	9.164	4.567
1500x1500	34.853	16.539
2000x2000	85.771	41.623
2500x2500	187.23	92.283
3000x3000	310.990	170.572

Параллельные методы сортировки. Пузырьковая сортировка

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, \dots, a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a_1', a_2', \dots, a_n') : a_1' \le a_2' \le \dots \le a_n'\}.$$

Возможные способы решения этой задачи широко обсуждаются в литературе; один из наиболее полных обзоров алгоритмов сортировки

содержится в работах. Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T \sim n^2.$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T \sim n\,log_2\,n$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких ($p > 1$) вычислительных элементов (процессоров или ядер). Исходный упорядочиваемый набор в этом случае «разделяется» на блоки, которые могут обрабатываться вычислительными элементами параллельно.

Анализ способов упорядочивания данных, применяемых в алгоритмах сортировки, позволяет судить о том, что многие методы основаны на применении одной и той же базовой операции "Сравнить и переставить", состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановки этих значений, если их порядок не соответствует условиям сортировки.

```
// базовая операция сортировки
if ( A[i] > A[j] ) {
  temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения собственно и проявляется различие алгоритмов сортировки.

Схема последовательного алгоритма пузырьковой сортировки

Последовательный алгоритм пузырьковой сортировки сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$$(a_1, a_2, \dots, a_n)$$

алгоритм сначала выполняет $n - 1$ базовых операций "сравнения-обмена" для последовательных пар элементов

$$(a_1, a_2), (a_2, a_3), \dots, (a_{n-1}, a_n).$$

В результате после первой итерации алгоритма самый большой элемент перемещается в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности

$$(a_1^{'}, a_{12}^{'}, \dots, a_{n-1}^{'}).$$

Как можно увидеть, последовательность будет отсортирована после $n - 1$ итерации.

Эффективность пузырьковой сортировки может быть улучшена, если завершать алгоритм в случае отсутствия каких-либо изменений сортируемой последовательности данных в ходе какой-либо итерации сортировки.

```
// последовательный аглгоритм пузырьковой сортировки
void BubbleSort(double *A, int n) {
  for (i = 0; i < n-1; i++)
    for (j = 0; j < n-1-i; j++)
      compare_exchange(A[j], A[j+1]);
}
```

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения обычно используется не сам этот алгоритм, а его модификация, известная в литературе как метод чет-нечетной перестановки. Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары

$$(a_1, a_2), (a_2, a_3), \dots, (a_{n-1}, a_n) \text{(при четном n)},$$

а на четных итерациях обрабатываются элементы

$$(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1}).$$

После n-кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

```
// последовательная реализация метода чет-нечетной
// перестановки
void OddEvenSort ( double* pData, int Size ) {
  double temp;
  int upper_bound;
  if (Size%2==0)
    upper_bound = Size/2-1;
  else
    upper_bound = Size/2;
  for (int i=0; i<Size; i++) {
    if (i%2 == 0) // четная итерация
      for (int j=0; j<Size/2; j++)
        compare_exchange(pData[2*j], pData[2*j+1]);
    else // нечетная итерация
      for (int j=0; j<upper_bound; j++)
        compare_exchange(pData[2*j+1], pData[2*j+2]);
  }
}
```

Схема параллельного алгоритма пузырьковой сортировки

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – несмотря на то, что четные и нечетные итерации должны выполняться строго последовательно, сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. Поскольку все вычислительные элементы имеют прямой доступ к каждому значению в сортируемом массиве, сравнение значений a_i и a_j может быть выполнено любым вычислительным элементом. При наличии нескольких вычислительных элементов появляется возможность одновременно выполнять операцию «Сравнить и переставить» над несколькими парами значений.

Рассмотрим возможный вариант реализации параллельного варианта метода пузырьковой сортировки. Используя OpenMP, получение параллельного алгоритма из последовательного достигается путем добавления двух директив препроцессора, при помощи которых распараллеливаются циклы сравнения пар значений на четных и нечетных итерациях метода чет-нечетной перестановки.

```
// параллельная реализация метода чет-нечетной
// перестановки
void ParallelOddEvenSort ( double* pData, int Size ) {
  double temp;
  int upper_bound;
  if (Size%2==0)
    upper_bound = Size/2-1;
  else
    upper_bound = Size/2;
  for (int i=0; i<Size; i++) {
    if (i%2 == 0) // четная итерация
      #pragma omp parallel for
        for (int j=0; j<Size/2; j++)
          compare_exchange(pData[2*j], pData[2*j+1]);
    else // // нечетная итерация
      #pragma omp parallel for
        for (int j=0; j<upper_bound; j++)
          compare_exchange(pData[2*j+1], pData[2*j+2]);
  }
}
```

В таблице 7 представлены результаты сравнения времени выполнения последовательного и параллельного (при распараллеливании на 4 потока) алгоритмов пузырьковой сортировки. Эксперименты проводились на вычислительном узле на базе четырехядерного процессора Intel(R) Core(TM) i5-3210M CPU @ 2.50ГГц, 4Гб RAM.

Таблица 7

Сравнительный анализ результатов работы программы

Размер массива	Время выполнения, сек.	
	последовательный алгоритм	параллельный алгоритм (4 потока)
10000	1.456	0.965
50000	36.277	21.035
100000	144.605	87.436

Задание

Реализовать и распараллелить с помощью OpenMP заданный алгоритм (согласно варианту).

Варианты заданий

№ варианта	Тип элемента данных	Алгоритм
1.	Длинное целое	Сортировка массива с помощью Шейкер-сортировки
2.	Знаковый короткий целый	Сортировка столбцов матрицы по возрастанию
3.	Беззнаковый целый	Сортировка массива методом пузырька
4.	Длинное целое	Сортировка массива методом QuickSort (быстрая сортировка)
5.	Целое	Сортировка массива методом Шелла.
6.	Беззнаковый целый	Сортировка массива с помощью сортировки вставками
7.	Длинное целое	Сортировка строк матрицы по возрастанию
8.	Знаковый короткий целый	Сортировка строк матрицы по убыванию
9.	Беззнаковый целый	Сортировка столбцов матрицы по убыванию
10.	Число с плавающей запятой	Перемножение матриц
11.	Длинное целое	Сортировка массива с помощью пирамидальной сортировки

ЛАБОРАТОРНАЯ РАБОТА №6.
OPENMP. ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ РЕШЕНИЯ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ В ЧАСТНЫХ ПРОИЗВОДНЫХ

Цель работы

Целью данной лабораторной работы является разработка параллельной программы, которая обеспечивает решение одной из задач, описываемой дифференциальным уравнением в частных производных – задачи Дирихле для уравнения Пуассона [11].

Задание

На основе многопоточного распараллеливания (технология OpenMP) выполнить высокопроизводительную реализацию сеточных алгоритмов решения краевых задач (задача Дирихле для уравнения Пуассона в квадратной области). Сравнить время выполнения и скорость сходимости (т.е. число итераций) параллельного алгоритма с непараллельными алгоритмами для различных значений параметра N (количество узлов сетки).

Информация

Краевая задача Дирихле для уравнения Пуассона:

$$\Delta u(x, y) = f(x, y), \cdots (x, y) \in G,$$
$$u(x, y) = g(x, y), \cdots (x, y) \in \partial G,$$

Здесь G – область в R^2, ∂G – ее граница, $\Delta = \partial^2/\partial x^2 + \partial^2/\partial y^2$ – двумерный оператор Лапласа. Функции f и g даны, требуется найти функцию u. Подобная краевая задача возникает при моделировании установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и т. д.

Для простоты будем рассматривать в качестве области G единичный квадрат:

$$G = \{(x, y) \in R^2 : 0 \le x \le 1, 0 \le y \le 1\}$$
$$G = \{(x,y) \in R^2 : 0 \le x \le 1, 0 \le y \le 1\}.$$

Функции f и g определим следующим образом:

$$f(x,y)=0, \quad (x,y) \in G,$$
$$g(x,0)=1-2x,$$
$$g(0,y)=1-2y,$$
$$g(x,1)=-1+2x,$$
$$g(1,y)=-1+2y.$$

При численном решении краевых задач, как правило, область G представляется в виде дискретной сетки узлов. Особенно просто подобная сетка определяется для прямоугольных областей. Например, для единичного квадрата равномерная сетка может быть описана следующим образом:

$$G_h = \{(x_i, y_j) \le R^2 : x_i = ih, y_j = jh, 0 \le i \le n + 1, 0 \le j \le n + 1, h = 1/(n + 1)\}.$$

Аппроксимацию функции $u(x, y)$ в точках x_i, y_j, будем обозначать через u_{ij}. Тогда, используя пятиточечный шаблон для вычисления оператора

Лапласа (рис.13), уравнение Пуассона можно представить в следующей конечно-разностной форме:

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

Здесь $f_{ij} = f(x_i, y_j)$. На рисунке 14 темные точки представляют внутренние узлы сетки (в которых требуется найти аппроксимацию функции u), а светлые точки граничные узлы, в которых значения функции u известны.

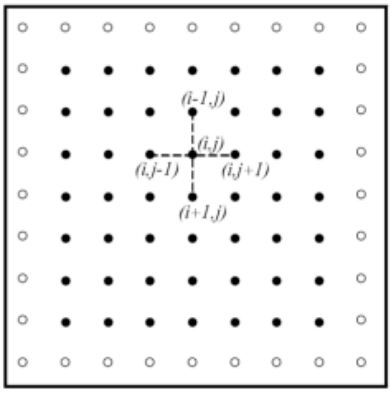


Рис. 14. Прямоугольная сетка в области D (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз)

Разностное уравнение можно разрешить относительно u_{ij}:

$$u_{ij} = 0{,}25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Полученное уравнение позволяет определить значение u_{ij}, по известным значениям аппроксимаций функции u в соседних узлах используемого шаблона. Это служит основой для построения различных итерационных схем решения задачи Дирихле. В подобных схемах вначале формируются некоторые приближения для значений u_{ij}, а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Примером итерационного метода является достаточно быстро сходящийся метод Гаусса-Зейделя, в котором при уточнении аппроксимаций используется следующая формула:

$$u_{ij}^{k} = 0{,}25(u_{i-1,j}^{k} + u_{i+1,j}^{k-1} + u_{i,j-1}^{k} + u_{i,j+1}^{k-1} - h^2 f_{ij}).$$

Следует обратить внимание на то, что в данном методе очередное, k - е приближение значения u_{ij} вычисляется по уже вычисленным на текущей (k - й) итерации значениям $u_{i-1,j}$ и $u_{i,j-1}$, в то время как в двух других узлах используются значения, найденные на предыдущей, $(k-1)$-й итерации. Итерационный процесс выполняется до тех пор, пока невязки $\left|u_{ij}^{k} - u_{ij}^{k-1}\right|$ во всех точках внутренней сетки не станут меньше некоторой заданной величины (требуемой точности вычислений).

Вариант непараллельной реализации алгоритма Гаусса-Зейделя

Листинг программы, реализующей непараллельный алгоритм Гаусса-Зейделя.

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include<omp.h>
#define OUTPUT 10
using namespace std;
double F(double x, double y)
{
  return 0;
}
double G(double x, double y)
{
  if (x == 0) return 1 - 2 * y;
  if (x == 1) return -1 + 2 * y;
  if (y == 0) return 1 - 2 * x;
  if (y == 1) return -1 + 2 * x;
  return 0;
}
void OutputFile(int N, double** u, string filename = "output.csv")
{
  ofstream file;
  file.open(filename);
  if(file.is_open()){
    file << fixed << setprecision(3);
    for (int i = 0; i < N + 2; i++)
    {
      for (int j = 0; j < N + 2; j++)
        file << setw(7) << u[i][j];
      file << endl;
    }
    file.close();
  }
}
void Output(int N, double** u)
{
  cout << fixed << setprecision(3);
  for (int i = 0; i < OUTPUT + 1; i++)
  {
    for (int j = 0; j < OUTPUT + 1; j++)
      cout << setw(7) << u[i*(N+1)/OUTPUT][j*(N+1)/OUTPUT] << ';';
    cout << endl;
  }
}
void Init(int N, double** u, double** f)
```

```
{
  double h = 1.0/(N+1);
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
      f[i][j] = F((i+1)*h, (j+1)*h);
    }
  for (int i = 1; i < N + 1; i++)
  {
    for (int j = 1; j < N+1; j++)
      u[i][j] = 0;
    u[i][0] = G(i*h, 0);
    u[i][N+1] = G(i*h, (N+1)*h);
  }
  for (int j = 0; j < N+2; j++)
  {
    u[0][j] = G(0,j*h);
    u[N+1][j] = G((N+1)*h,j*h);
  }
  Output(N,u);
}
void Calc(int N, double** u, double** f, double eps, int& IterCnt)
{
  double max;
  double h = 1.0/(N+1);
  do
  {
    IterCnt++;
    max = 0;
    for (int i = 1; i < N+1; i++)
      for (int j = 1; j < N+1; j++)
      {
        double u0 = u[i][j];
        u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - h*h*f[i-
1][j-1]);
        double d = abs(u[i][j]-u0);
        if (d > max)
          max = d;
      }
  }
  while (max > eps);
  cout << endl << " Max error: " << setprecision(7) << max << endl;
}
int main()
{
```

```
    int N, IterCnt = 0;
    double eps;
    N = 199;
    eps = 0.000001;
    double **u = new double*[N+2], **f = new double*[N];
    for (int i = 0; i < N; i++)
      f[i] = new double[N];
    for (int i = 0; i < N + 2; i++)
      u[i] = new double[N+2];
    cout<<endl;
    Init(N,u,f);
    double tmbeg = omp_get_wtime(), tmend;
    Calc(N,u,f,eps,IterCnt);
    tmend = omp_get_wtime();
    cout << " Time = " << setprecision(6) << tmend - tmbeg << " IterCnt = "
<< IterCnt << endl << endl;
    Output(N,u);
    OutputFile(N,u,"output_all.csv");
    for (int i = 0; i < N; i++)
      delete[] f[i];
    delete[] f;
    for (int i = 0; i < N + 2; i++)
      delete[] u[i];
    delete[] u;
    cin.get();
    return 0;
  }
```

Для хранения приближенного решения функции *u* (и дискретизации функции *f*) используются двумерные массивы.

Выводятся не все найденные значения дискретизаций, а *OUTPUT*=10 строк и столбцов массива, включая начальную и конечную строку и столбец.

Вывод выполняется в консольное окно; помимо результатов решения выводятся следующие данные: максимальная ошибка (невязка) среди всех точек внутренней сетки, количество итераций, время, потребовавшееся для вычисления массива *u* с требуемой точностью *eps = 0.000001*. Также предусмотрен вывод всего массива *u* с найденным решением в csv-файл.

Приведем текст, выведенный в консольном окне (рис. 15):

```
 1.000  0.800  0.600  0.400  0.200  0.000 -0.200 -0.400 -0.600 -0.800 -1.000
 0.800  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.800
 0.600  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.600
 0.400  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.400
 0.200  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.200
 0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
-0.200  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.200
-0.400  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.400
-0.600  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.600
-0.800  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.800
-1.000 -0.800 -0.600 -0.400 -0.200  0.000  0.200  0.400  0.600  0.800  1.000

Max error: 0.0000010
Time = 12.869869 IterCnt = 6208

 1.000  0.800  0.600  0.400  0.200  0.000 -0.200 -0.400 -0.600 -0.800 -1.000
 0.800  0.640  0.480  0.320  0.160  0.000 -0.160 -0.319 -0.479 -0.640 -0.800
 0.600  0.480  0.359  0.239  0.120  0.000 -0.119 -0.239 -0.359 -0.480 -0.600
 0.400  0.320  0.239  0.159  0.080  0.000 -0.079 -0.159 -0.239 -0.320 -0.400
 0.200  0.160  0.120  0.080  0.040  0.000 -0.040 -0.080 -0.120 -0.160 -0.200
 0.000  0.000  0.000  0.000  0.000 -0.000 -0.000 -0.000 -0.000 -0.000  0.000
-0.200 -0.160 -0.119 -0.079 -0.040 -0.000  0.039  0.079  0.119  0.160  0.200
-0.400 -0.319 -0.239 -0.159 -0.080 -0.000  0.079  0.159  0.239  0.319  0.400
-0.600 -0.479 -0.359 -0.239 -0.120 -0.000  0.119  0.239  0.359  0.479  0.600
-0.800 -0.640 -0.480 -0.320 -0.160 -0.000  0.160  0.319  0.479  0.640  0.800
-1.000 -0.800 -0.600 -0.400 -0.200  0.000  0.200  0.400  0.600  0.800  1.000
```

Рис. 15. Результаты непараллельной реализации решения задачи Дирихле для уравнения Пуассона

График решения u(x, y) в данном случае будет представлять собой гиперболический параболоид (седло). На рисунке 16 представлена графическая иллюстрация полученного приближенного решения.

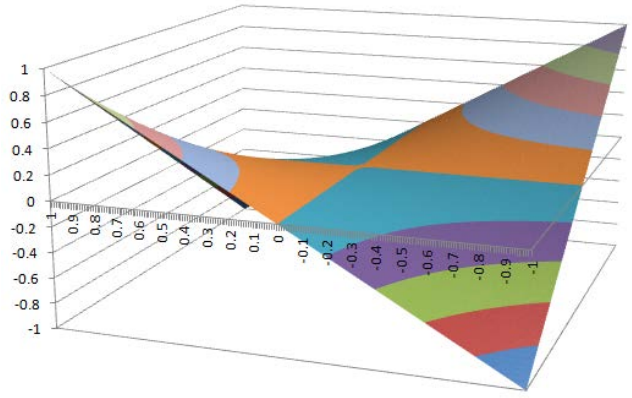


Рис.16. Графическая иллюстрация полученного приближенного решения задачи Дирихле для уравнения Пуассона

Вариант распараллеливания с применением OpenMP

Реализация распараллеливания алгоритма Гаусса-Зейделя при решении задачи Дирихле для уравнения Пуассона представлена за счет модификации функции Calc() в виде функции OMPCalc(). В представленной реализации распараллеливание выполняется на 4 потока.

```
void OMPCalc(int N, double** u, double** f, double eps, int& IterCnt)
{
  double max;
  double h = 1.0/(N+1);
  do
  {
```

```
        IterCnt++;
        max = 0;
        #pragma omp parallel for shared(u,N,max) num_threads(4)
        for (int i = 1; i < N+1; i++)
        {
          double max0 = 0;
          for (int j = 1; j < N+1; j++)
          {
            double u0 = u[i][j];
            u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - h*h*f[i-
1][j-1]);
            double d = abs(u[i][j]-u0);
            if (d > max0)
              max0 = d;
          }
          if (max0 > max)
            #pragma omp critical
              if (max0 > max)
                max = max0;
        }
      }
      while (max > eps);
      cout << endl << " Max error: " << setprecision(7) << max << endl;
    }
```

На рисунке 17 представлены результаты параллельной реализации решения задачи Дирихле для уравнения Пуассона.

```
 1.000  0.800  0.600  0.400  0.200  0.000 -0.200 -0.400 -0.600 -0.800 -1.000
 0.800  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.800
 0.600  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.600
 0.400  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.400
 0.200  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 -0.200
 0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
-0.200  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.200
-0.400  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.400
-0.600  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.600
-0.800  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.800
-1.000 -0.800 -0.600 -0.400 -0.200  0.000  0.200  0.400  0.600  0.800  1.000

Max error: 0.0000010
Time = 5.727607 IterCnt = 6206

 1.000  0.800  0.600  0.400  0.200  0.000 -0.200 -0.400 -0.600 -0.800 -1.000
 0.800  0.640  0.480  0.320  0.160  0.000 -0.160 -0.319 -0.479 -0.640 -0.800
 0.600  0.480  0.359  0.239  0.120  0.000 -0.119 -0.239 -0.359 -0.480 -0.600
 0.400  0.320  0.239  0.159  0.080  0.000 -0.079 -0.159 -0.239 -0.320 -0.400
 0.200  0.160  0.120  0.080  0.040  0.000 -0.040 -0.080 -0.120 -0.160 -0.200
 0.000  0.000  0.000  0.000  0.000 -0.000 -0.000 -0.000 -0.000 -0.000  0.000
-0.200 -0.160 -0.119 -0.079 -0.040 -0.000  0.039  0.079  0.119  0.160  0.200
-0.400 -0.319 -0.239 -0.159 -0.080 -0.000  0.079  0.159  0.239  0.319  0.400
-0.600 -0.479 -0.359 -0.239 -0.120 -0.000  0.119  0.239  0.359  0.479  0.600
-0.800 -0.640 -0.480 -0.320 -0.160 -0.000  0.160  0.319  0.479  0.640  0.800
-1.000 -0.800 -0.600 -0.400 -0.200  0.000  0.200  0.400  0.600  0.800  1.000
```

Рис. 17 . Результаты параллельной реализации (num_threads = 4) решения задачи Дирихле для уравнения Пуассона

На рисунке 18 представлен сравнительный анализ результатов распараллеливания на несколько потоков алгоритма Гаусса-Зейделя при решении задачи Дирихле для уравнения Пуассона.

```
OpenMP (num_threads = 1):
Max error: 9.995984e-007 Time = 12.6677 IterCnt = 6208
OpenMP (num_threads = 2):
Max error: 9.997752e-007 Time = 6.90058 IterCnt = 6207
OpenMP (num_threads = 4):
Max error: 9.999332e-007 Time = 6.0031 IterCnt = 6205
OpenMP (num_threads = 8):
Max error: 9.987191e-007 Time = 8.80898 IterCnt = 6222
OpenMP (num_threads = 16):
Max error: 9.920634e-007 Time = 10.1584 IterCnt = 6271
OpenMP (num_threads = 64):
Max error: 9.988492e-007 Time = 12.7233 IterCnt = 6349
```

Рис. 18. Сравнительный анализ результатов распараллеливания на несколько потоков (1, 2, 4, 8, 16, 64 потоков) алгоритма Гаусса-Зейделя

Индивидуальные задания для выполнения:

На основе многопоточного распараллеливания (технология OpenMP) выполнить высокопроизводительную реализацию сеточных алгоритмов решения краевых задач.

Варианты заданий:

1. Апельсины в течение короткого времени могут выдерживать отрицательные температуры. Предположим, что апельсин диаметром 0,1 м (□=0,47 Вт/(м·град), c=3800 Дж/(кг·град), ρ=940 кг/м^3) имеет начальную температуру +5 °C. Температура воздуха внезапно падает до –5 °C. Построить математическую модель для определения момента времени, когда температура поверхности апельсина достигнет 0 °C. Коэффициент теплоотдачи от апельсина к воздуху равен 10 Вт/(м^2·град). Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различной размерности задачи и числа используемых процессов (потоков).

2. Начальная температура хлорвинилового шарика (λ=0,15 Вт/(м·град), α=8·10^{-8} м^2/с) диаметром 5 см равна 90 °C. Он погружается в бак с водой, имеющей температуру 20 °C. Коэффициент теплоотдачи от шарика в воде 20 Вт/(м^2·град). Построить математическую модель процесса охлаждения шарика. Найти время пребывания шарика в воде, по истечении которого температура в его центре достигнет 40 °C. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различных размеров вычислительной сетки и числа используемых процессов (потоков).

3. Длинный алюминиевый (λ=236 Вт/(м·град), α =10-4 м^2/с) цилиндр диаметром 0,6 м имеет начальную температуру 200 °C. Его внезапно помещают в среду с температурой 70 °C и коэффициентом теплоотдачи 85 Вт/(м^2·град). Построить математическую модель процесса охлаждения цилиндра. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различных размеров сетки и числа используемых процессов (потоков).

4. Лист оконного стекла имеет толщину 4 мм. Температура одной поверхности 0 °C, а другой – +20 °C. Найти распределение температуры по

толщине стекла через заданное время, если температура за окном упала до – 10 °С. Записать математическую постановку задачи. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различной плотности узлов вычислительной сетки и числа используемых процессов (потоков).

5. Стенка большой печи толщиной 1,5 см изготовлена из чугуна (λ =83,5 Вт/(м·град), α = 22·10-6 м^2/с). Температура горячего газа 1100 °С. Коэффициент конвективной теплоотдачи на внутренней поверхности стенки 250 Вт/(м^2·град). Наружная поверхность печи окружена воздухом с температурой 30 °С и коэффициентом теплоотдачи 20 Вт/(м^2·град). Записать постановку задачи теплопроводности для стенки и найти распределение температур в ней. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различного числа используемых процессов (потоков).

6. Определить конечное распределение температуры в обручальном кольце с радиусами R=2,0 см и r=1,8 см, если температура в помещении 20 °С, а температуру тела человека можно принять равной 36,6 °С. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различных размеров вычислительной сетки и числа используемых процессов (потоков). α =126·10^{-6} м^2/с.

7. Бетонный цилиндр диаметром 10 см и длиной 2,5 м имеет начальную температуру 90 °С. Он охлаждается в воздухе при температуре 10 °С. Коэффициент теплоотдачи от бетона к воздуху 18 Вт/(м^2·град). Найти распределение температур в цилиндре с течением времени. Определить время, за которое температура в центре цилиндра достигнет 30 °С. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование.

8. Нужно нагреть кусок алюминиевой проволоки, пропуская по ней электрический ток. Диаметр проволоки 1 мм, длина 10 см, электрическое сопротивление 0,2 Ом. По ней пропускается постоянный ток силой 1 А в течение 60 с. Начальная температура проволоки 25 °С. Проволока находится в воздухе с температурой 25 °С и коэффициентом теплоотдачи 20 Вт/(м^2·град). Найти распределение температуры в проволоке с течением времени. Записать математическую формулировку задачи. Пояснение: объемная интенсивность внутренних источников тепловыделения при пропускании электрического тока = сила тока·сила тока·омическое сопротивление/объем. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование.

9. Электродетонатор имеет форму цилиндра диаметром 0,1 мм и длиной 5 мм. Он находится в воздухе с температурой 30 °С и коэффициентом теплоотдачи 10 Вт/(м^2·град). Теплофизические свойства детонатора: λ=20 Вт/(м·град), α =5·10-5 м^2/с, электрическое сопротивление 0,2 Ом. Пренебрегая излучением и утечками тепла в креплениях на концах детонатора, определить время, по истечении которого детонатор взорвется, если по детонатору пропускать постоянный ток силой 3 А. Пояснение: объемная интенсивность

внутренних источников тепловыделения при пропускании электрического тока = сила тока·сила тока·омическое сопротивление/объем. Температура плавления материала детонатора 900 °С. Записать математическую постановку задачи. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различной размерности задачи и числа используемых процессов (потоков).

10. Начальная температура длинной стальной балки с сечением квадрата единичной длины равна 40 °С. Определить распределение температуры тела через 100 с, если на боковых границах задать следующие граничные условия: при X=0:T=20; при X=1:T=10; при Y=0:T=10; при Y=1:T=10. Записать математическую постановку задачи. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование.

Контрольные вопросы

1. Приведите классификацию современных высокопроизводитель-ных средств вычислительной техники?
2. Какие компьютерные платформы относятся к числу вычислительных систем с общей памятью?
3. Какие существуют методики для оценки производительности вычислительных систем?
4. Какие подходы используются для разработки параллельных программ?
5. Перечислите основные функции для работы с потоками в WinAPI?
6. Перечислите способы завершения потока в WinAPI. Какой из них является наиболее безопасным?
7. Перечислите основные базовые классы приоритетов потоков в WinAPI?
8. Приведите классификацию средств синхронизации в WinAPI?
9. В чем состоят основы и преимущества технологии OpenMP?
10. Что понимается под параллельной программой в рамках технологии OpenMP?
11. Какой формат записи директив OpenMP?
12. В чем состоит понятие фрагмента, области и секции параллельной программы?
13. Какой минимальный набор директив OpenMP позволяет начать разработку параллельных программ?
14. Как определить время выполнения OpenMP программы?
15. Как осуществляется распараллеливание циклов в OpenMP?
16. Какие возможности имеются в OpenMP для управления распределением итераций циклов между потоками?
17. Как определяется порядок выполнения итераций в распараллеливаемых циклах в OpenMP?
18. Какие правила синхронизации вычислений в распараллеливаемых циклах в OpenMP?
19. Как можно ограничить распараллеливание фрагментов программного кода с невысокой вычислительной сложностью?
20. Как определяются общие и локальные переменные потоков?
21. Что понимается под операцией редукции?
22. Какие способы организации взаимоисключения могут быть использованы в OpenMP?
23. Что понимается под атомарной (неделимой) операцией?
24. Какие проблемы синхронизации могут возникнуть при многопоточной обработке?

ЗАКЛЮЧЕНИЕ

Данное учебное издание представляет собой практикум по дисциплине «Методы и средства высокопроизводительного программирования» для магистрантов направления 09.04.01 – «Информатика и вычислительная техника».

В практикуме рассматриваются: классификация, архитектура современных вычислительных систем и методики оценки их производительности, подход в организации параллельных вычислений, реализуемых средствами Windows API; технология OpenMP, как одна из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования.

Описание функциональности WinAPI и OpenMP при организации высокопроизводительных вычислений в данном практикуме сопровождается примерами и лабораторными работами для самостоятельного изучения и выполнения студентами на практике.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Атли, К. Visual Basic. NET для программистов [Электронный ресурс] / К. Атли; Пер. с англ. - Москва: ДМК Пресс, 2008. - 304 с.

2. Дадян, Э. Г. Современные технологии программирования. Язык C# : учебник : в 2 томах. Том 2. Для продвинутых пользователей / Э.Г. Дадян. — Москва: ИНФРА-М, 2021 – 335 с.

3. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. — СПб.: Питер, 2003. – 461 с.

4. MSDN Library. [Электронный ресурс] – Режим доступа к ресурсу: http://www.msdn.microsoft.com/en-us/library/.

5. Баденко В. Л. Высокопроизводительные вычисления: учеб. пособие – СПб.: Изд-во Политехн. ун-та, 2010. – 180 с.

6. 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.; Издательство «Русская редакция»; 2008. – 720 стр.

7. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2002. – 544 с.

8. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

9. Гергель В.П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем. Учебное пособие – Нижний Новгород; Изд-во ННГУ им. Н.И.Лобачевского, 2010. – 421 с.

10. Кнут Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. 2 издание - М.: Издательский дом "Вильямс", 2018. – 834 с.

11. Кузнецов Г.В., Шеремет М.А. Разностные методы решения задач теплопроводности: учебное пособие. / Г.В. Кузнецов, М.А. Шеремет. – Томск: Изд-во ТПУ, 2007. – 172 с.

OpenMP. The OpenMP API specification for parallel programming [Электронный ресурс]. URL: https://www.openmp.org/specifications/ (дата обращения: 01.06.2022).

Учебное текстовое электронное издание

Калитаев Александр Николаевич
Егорова Людмила Геннадьевна
Торчинский Вадим Ефимович

МЕТОДЫ И СРЕДСТВА ВЫСОКОПРОИЗВОДИТЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Практикум

Ответственность за содержание возлагается на авторов
Издается полностью в авторской редакции

0,81 Мб
1 электрон. опт. диск

г. Магнитогорск, 2022 год
ФГБОУ ВО «МГТУ им. Г.И. Носова»
Адрес: 455000, Россия, Челябинская область, г. Магнитогорск,
пр. Ленина 38

ФГБОУ ВО «Магнитогорский государственный
технический университет им. Г.И. Носова»
Кафедра вычислительной техники и программирования
Библиотечно-информационный комплекс
e-mail: bik@magtu.ru