

OpenMP

OpenMP (Open Multi-Processing) - это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью (SMP-системах).

Первый стандарт OpenMP был разработан в 1997 г. как API, ориентированный на написание легко переносимых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и языки Си и Си++.

Интерфейс OpenMP стал одной из наиболее популярных технологий параллельного программирования. OpenMP успешно используется как при программировании суперкомпьютерных систем с большим количеством процессоров, так и в настольных пользовательских системах или, например, в Xbox 360.

Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения, чья работа регулируется некоммерческой организацией "OpenMP Architecture Review Board" (ARB) [1].

В OpenMP используется модель параллельного выполнения "ветвление-слияние". Программа OpenMP начинается как единственный поток выполнения, называемый начальным потоком. Когда поток встречает параллельную конструкцию, он создает новую группу потоков, состоящую из себя и некоторого числа дополнительных потоков, и становится главным в новой группе. Все члены новой группы (включая главный) выполняют код внутри параллельной конструкции. В конце параллельной конструкции имеется неявный барьер. После параллельной конструкции выполнение пользовательского кода продолжает только главный поток. В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков. Вложенные регионы могут в свою очередь включать регионы более глубокого уровня вложенности.

Число потоков в группе, выполняющихся параллельно, можно контролировать несколькими способами. Один из них - использование переменной окружения OMP_NUM_THREADS. Другой способ - вызов процедуры `omp_set_num_threads()`. Еще один способ - использование выражения `num_threads` в сочетании с директивой `parallel`.

OpenMP и другие технологии параллельного программирования

На данный момент считается, что наиболее гибким, переносимым и общепринятым интерфейсом параллельного программирования является MPI (интерфейс передачи сообщений). Однако модель передачи сообщений:

- недостаточно эффективна на SMP-системах;
- относительно сложна в освоении, так как требует мышления в "невыхислительных" терминах.

POSIX-интерфейс для организации нитей (Pthreads) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования:

- нет поддержки Fortran;
- слишком низкий уровень;
- нет поддержки параллелизма по данным;

- механизм нитей изначально разрабатывался не для целей организации вычислительного параллелизма.

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей). Перечислим преимущества, которые OpenMP дает разработчику.

За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.

При этом OpenMP - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.

Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки (stubs), текст которых приведен в спецификациях.

Одним из достоинств OpenMP его разработчики считают поддержку так называемых "orphan" (оторванных) директив, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.


OpenMP и инструментарий

На данный момент технология OpenMP поддерживается большинством компиляторов языка Си/Си++. Несколько хуже дело обстоит с инструментами тестирования параллельных OpenMP программ. Инструменты анализа, проверки и оптимизации параллельных программ хотя и существуют давно, до недавнего времени были мало востребованы при разработке прикладного программного обеспечения. Поэтому они часто являются менее удобными, чем иные инструментальные средства разработки.

Наиболее полно процесс разработки параллельных OpenMP программ поддержан в пакете Intel Parallel Studio. Имеется инструмент предварительного анализа кода, для выявления участков кода, которые потенциально можно эффективно распараллелить. Имеется хорошо оптимизирующий компилятор с поддержкой OpenMP. Имеется профилировщик и инструмент динамического анализа для выявления параллельных ошибок.

Дополнительно можно выделить инструмент VivaMP, входящий в состав PVS-Studio. Это статический анализатор кода, специализированный на выявлении ошибок в OpenMP программах на этапе их написания.

Реализация многопоточности без лишних усилий

Среди специалистов, занимающихся параллельными вычислениями, популярна шутка «Параллельные вычисления — технология будущего...  и так будет всегда». Эта шутка не теряет актуальность уже несколько десятилетий. Аналогичные настроения были распространены в сообществе разработчиков архитектур компьютеров, обеспокоенном тем, что скоро будет достигнут предел тактовой частоты процессоров, однако частоты процессоров продолжают повышаться, хотя гораздо медленнее, чем раньше. Сплав оптимизма специалистов по параллельным вычислениям и пессимизма архитекторов систем способствовал появлению революционных многоядерных процессоров.

Главные производители процессоров сместили акцент с повышения тактовых частот на реализацию параллелизма в самих процессорах за счет использования многоядерной архитектуры. Идея проста: интегрировать в один процессор более одного ядра. Система, включающая процессор с двумя ядрами, по сути, не отличается от двухпроцессорного компьютера, а система с четырехядерным процессором — от четырехпроцессорного. Этот подход позволяет избежать многих технологических проблем, связанных с повышением тактовых частот, и создавать при этом более производительные процессоры.

Все это прекрасно, но если ваше приложение не будет использовать несколько ядер, его быстродействие никак не изменится. Именно здесь и вступает в игру технология OpenMP, которая помогает программистам на C++ быстрее создавать многопоточные приложения.

Подробно описать OpenMP в одной статье просто невозможно, так как это очень объемный и мощный API. Рассматривайте эту статью как введение, где демонстрируется применение различных средств OpenMP для быстрого написания многопоточных программ. Если вам понадобится дополнительная информация по этой тематике, мы рекомендуем обратиться к спецификации, доступной на сайте OpenMP (www.openmp.org), — она на удивление легко читается.

Активизация OpenMP в Visual C++

Стандарт OpenMP был разработан в 1997 г. как API, ориентированный на написание портируемых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и C/C++. Последняя версия OpenMP — 2.0; ее полностью поддерживает Visual C++ 2005. Стандарт OpenMP поддерживается и платформой Xbox 360.

Прежде чем заниматься кодом, вы должны знать, как активизировать реализованные в компиляторе средства OpenMP. Для этого служит появившийся в Visual C++ 2005 параметр компилятора `/openmp`. (Вы можете активизировать директивы OpenMP на страницах свойств проекта, выбрав Configuration Properties, C/C++, Language и изменив значение свойства OpenMP Support.) Встретив параметр `/openmp`, компилятор определяет символ `_OPENMP`, с помощью которого можно выяснить, включены ли средства OpenMP. Для этого достаточно написать `#ifndef _OPENMP`.

OpenMP связывается с приложениями через библиотеку импорта `vcomp.lib`. Соответствующая библиотека периода выполнения называется `vcomp.dll`. Отладочные версии библиотек импорта и периода выполнения (`vcompd.lib` и `vcompd.dll` соответственно) поддерживают дополнительные сообщения об ошибках, генерируемых при некоторых недопустимых операциях. Имейте в виду, что Visual C++ не поддерживает статическое связывание с библиотекой OpenMP периода выполнения, хотя в версии для Xbox 360 это поддерживается.

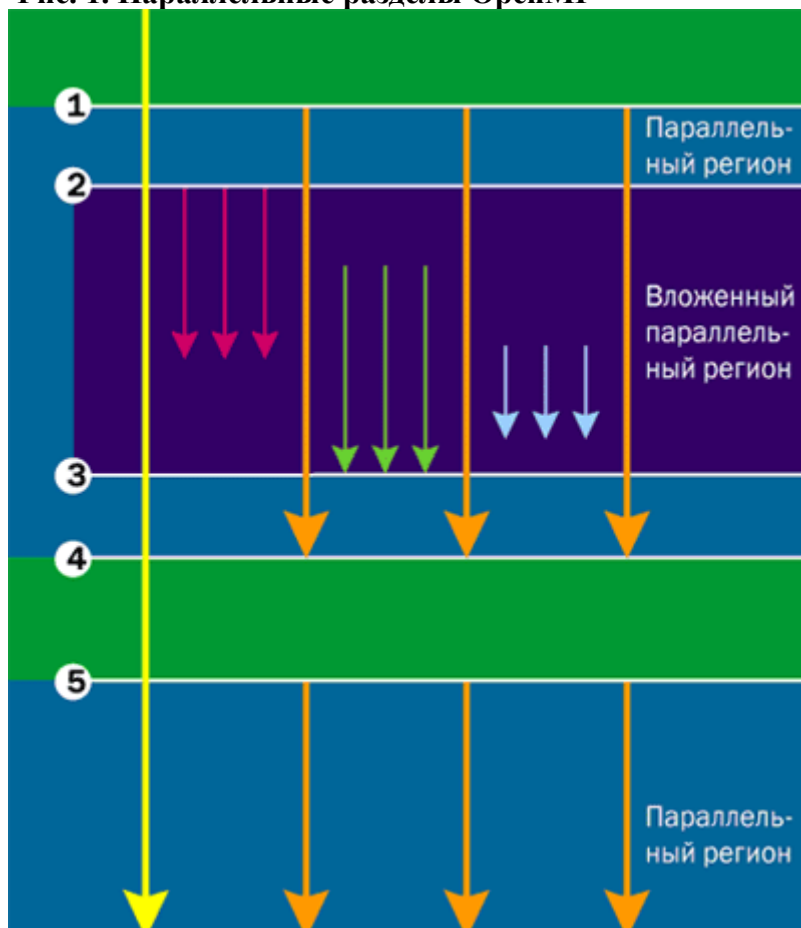
Параллельная обработка в OpenMP

Работа OpenMP-приложения начинается с единственного потока — основного. В приложении могут содержаться параллельные регионы, входя в которые, основной поток создает группы потоков (включающие основной поток). В конце параллельного региона группы потоков останавливаются, а выполнение основного потока продолжается. В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков.

Вложенные регионы могут в свою очередь включать регионы более глубокого уровня вложенности.

Параллельную обработку в OpenMP иллюстрирует рис. 1. Самая левая стрелка представляет основной поток, который выполняется в одиночестве, пока не достигает первого параллельного региона в точке 1. В этой точке основной поток создает группу потоков, и теперь все они одновременно выполняются в параллельном регионе.

Рис. 1. Параллельные разделы OpenMP



В точке 2 три из этих четырех потоков, достигнув вложенного параллельного региона, создают новые группы потоков. Исходный основной и потоки, создавшие новые группы, становятся владельцами своих групп (основными в этих группах). Учтите, что потоки могут создавать новые группы в разные моменты или вообще не встретить вложенный параллельный регион.

В точке 3 вложенный параллельный регион завершается. Каждый поток вложенного параллельного региона синхронизирует свое состояние с другими потоками в этом регионе, но синхронизация разных регионов между собой не выполняется. В точке 4 заканчивается первый параллельный регион, а в точке 5 начинается новый. Локальные данные каждого потока в промежутках между параллельными регионами сохраняются.

Таковы основы модели выполнения в OpenMP. Теперь вы готовы узнать, с чего начать разработку параллельного приложения.

Конструкции OpenMP

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы `pragma` и функции исполняющей среды OpenMP. Директивы `pragma`, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с `#pragma omp`. Как и любые другие директивы `pragma`, они игнорируются компилятором, не поддерживающим конкретную технологию — в данном случае OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API-функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл `omp.h`. Если вы используете в приложении только OpenMP-директивы `pragma`, включать этот файл не требуется.

Для реализации параллельного выполнения блоков приложения нужно просто добавить в код директивы `pragma` и, если нужно, воспользоваться функциями библиотеки OpenMP периода выполнения. Директивы `pragma` имеют следующий формат:

```
#pragma omp <директива> [раздел [ [,] раздел]...]
```

OpenMP поддерживает директивы `parallel`, `for`, `parallel for`, `section`, `sections`, `single`, `master`, `critical`, `flush`, `ordered` и `atomic`, которые определяют или механизмы разделения работы или конструкции синхронизации. В этой статье мы обсудим большинство директив.

Раздел (clause) — это необязательный модификатор директивы, влияющий на ее поведение. Списки разделов, поддерживаемые каждой директивой, различаются, а пять директив (`master`, `critical`, `flush`, `ordered` и `atomic`) вообще не поддерживают разделы.

Реализация параллельной обработки

Хотя директив OpenMP много, все они сразу нам не понадобятся. Самая важная и распространенная директива — `parallel`. Она создает параллельный регион для следующего за ней структурированного блока, например:

```
#pragma omp parallel [раздел[ [,] раздел]...]  
структурированный блок
```

Эта директива сообщает компилятору, что структурированный блок кода должен быть выполнен параллельно, в нескольких потоках. Каждый поток будет выполнять один и тот же поток команд, но не один и тот же набор команд — все зависит от операторов, управляющих логикой программы, таких как `if-else`.

В качестве примера рассмотрим классическую программу «Hello World»:

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

В двухпроцессорной системе вы, конечно же, рассчитывали бы получить следующее:

```
Hello World  
Hello World
```

Тем не менее, результат мог бы оказаться и таким:
HellHell oo WorWlodrl
d

Второй вариант возможен из-за того, что два выполняемых параллельно потока могут попытаться вывести строку одновременно. Когда два или более потоков одновременно пытаются прочитать или изменить общий ресурс (в нашем случае им является окно консоли), возникает вероятность гонок (race condition). Это недетерминированные ошибки в коде программы, найти которые крайне трудно. За предотвращение гонок отвечает программист; как правило, для этого используют блокировки или сводят к минимуму обращения к общим ресурсам.

Давайте взглянем на более серьезный пример, который определяет средние значения двух соседних элементов массива и записывает результаты в другой массив. В этом примере используется новая для вас OpenMP-конструкция `#pragma omp for`, которая относится к директивам разделения работы (work-sharing directive). Такие директивы применяются не для параллельного выполнения кода, а для логического распределения группы потоков, чтобы реализовать указанные конструкции управляющей логики. Директива `#pragma omp for` сообщает, что при выполнении цикла `for` в параллельном регионе итерации цикла должны быть распределены между потоками группы:

```
#pragma omp parallel
{
#pragma omp for
for(int i = 1; i < size; ++i)
    x[i] = (y[i-1] + y[i+1])/2;
}
```

Если бы этот код выполнялся на четырехпроцессорном компьютере, а у переменной `size` было бы значение 100, то выполнение итераций 1—25 могло бы быть поручено первому процессору, 26—50 — второму, 51—75 — третьему, а 76—99 — четвертому. Это характерно для политики планирования, называемой статической. Политики планирования мы обсудим позднее.

Следует отметить, что в конце параллельного региона выполняется барьерная синхронизация (barrier synchronization). Иначе говоря, достигнув конца региона, все потоки блокируются до тех пор, пока последний поток не завершит свою работу.

Если из только что приведенного примера исключить директиву `#pragma omp for`, каждый поток выполнит полный цикл `for`, проделав много лишней работы:

```
#pragma omp parallel
{
for(int i = 1; i < size; ++i)
    x[i] = (y[i-1] + y[i+1])/2;
}
```

Так как циклы являются самыми распространенными конструкциями, где выполнение кода можно распараллелить, OpenMP поддерживает сокращенный способ записи комбинации директив `#pragma omp parallel` и `#pragma omp for`:

```
#pragma omp parallel for
for(int i = 1; i < size; ++i)
    x[i] = (y[i-1] + y[i+1])/2;
```

Обратите внимание, что в этом цикле нет зависимостей, т. е. одна итерация цикла не зависит от результатов выполнения других итераций. А вот в двух следующих циклах есть два вида зависимости:

```
for(int i = 1; i <= n; ++i) // цикл 1
    a[i] = a[i-1] + b[i];
```

```
for(int i = 0; i < n; ++i) // цикл 2
    x[i] = x[i+1] + b[i];
```

Распараллелить цикл 1 проблематично потому, что для выполнения итерации i нужно знать результат итерации $i-1$, т. е. итерация i зависит от итерации $i-1$. Распараллелить цикл 2 тоже проблематично, но по другой причине. В этом цикле вы можете вычислить значение $x[i]$ до $x[i-1]$, однако, сделав так, вы больше не сможете вычислить значение $x[i-1]$. Наблюдается зависимость итерации $i-1$ от итерации i .

При распараллеливании циклов вы должны убедиться в том, что итерации цикла не имеют зависимостей. Если цикл не содержит зависимостей, компилятор может выполнять цикл в любом порядке, даже параллельно. Соблюдение этого важного требования компилятор не проверяет — вы сами должны заботиться об этом. Если вы укажете компилятору распараллелить цикл, содержащий зависимости, компилятор подчинится, что приведет к ошибке.

Кроме того, OpenMP налагает ограничения на циклы `for`, которые могут быть включены в блок `#pragma omp for` или `#pragma omp parallel for block`. Циклы `for` должны соответствовать следующему формату:

```
for([целочисленный тип] i = инвариант цикла;
    i {<, >, =, <=, >=} инвариант цикла;
    i {+, -} = инвариант цикла)
```

Эти требования введены для того, чтобы OpenMP мог при входе в цикл определить число итераций.

Сравнение поддержки потоков в OpenMP и Win32

Думаем, будет полезно сравнить только что приведенный пример, включающий директиву `#pragma omp parallel for`, с кодом, который пришлось бы написать для решения той же задачи на основе Windows API. Как видно в листинге 1, для достижения того же результата требуется гораздо больше кода, а за кулисами в этом варианте выполняются еще кое-какие операции. Так, конструктор класса `ThreadData` определяет, какими должны быть значения `start` и `stop` при каждом вызове потока. OpenMP обрабатывает все эти детали сам и предоставляет программисту дополнительные средства конфигурирования параллельных регионов и кода.

Листинг 1. Многопоточность в Win32

```
class ThreadData {
public:
    // Конструктор инициализирует поля start и stop
    ThreadData(int threadNum);
    int start;
    int stop;
```

```

};

DWORD ThreadFn(void* passedInData)
{
    ThreadData *threadData = (ThreadData *)passedInData;
    for(int i = threadData->start; i < threadData->stop; ++i )
        x[i] = (y[i-1] + y[i+1]) / 2;
    return 0;
}

void ParallelFor()
{
    // Запуск групп потоков
    for(int i=0; i < nTeams; ++i)
        ResumeThread(hTeams[i]);

    // Для каждого потока здесь неявно вызывается
    // метод ThreadFn

    // Ожидание завершения работы
    WaitForMultipleObjects(nTeams, hTeams, TRUE, INFINITE);
}

int main(int argc, char* argv[])
{
    // Создание групп потоков
    for(int i=0; i < nTeams; ++i)
    {
        ThreadData *threadData = new ThreadData(i);
        hTeams[i] = CreateThread(NULL, 0, ThreadFn, threadData,
            CREATE_SUSPENDED, NULL);
    }

    ParallelFor(); // имитация OpenMP-конструкции parallel for

    // Очистка
    for(int i=0; i < nTeams; ++i)
        CloseHandle(hTeams[i]);
}

```

Общие и частные данные

Разрабатывая параллельные программы, вы должны понимать, какие данные являются общими (shared), а какие частными (private), — от этого зависит не только производительность, но и корректная работа программы. В OpenMP это различие очевидно, к тому же вы можете настроить его вручную.

Общие переменные доступны всем потокам из группы, поэтому изменения таких переменных в одном потоке видимы другим потокам в параллельном регионе. Что касается частных переменных, то каждый поток из группы располагает их отдельными экземплярами, поэтому изменения таких переменных в одном потоке никак не сказываются на их экземплярах, принадлежащих другим потокам.

По умолчанию все переменные в параллельном регионе — общие, но из этого правила есть три исключения. Во-первых, частными являются индексы параллельных циклов `for`. Например, это относится к переменной `i` в коде, показанном в листинге 2. Переменная `j` по умолчанию не является частной, но явно сделана таковой через раздел `firstprivate`.

Листинг 2. Разделы директив OpenMP и вложенный цикл `for`

```
float sum = 10.0f;
MatrixClass myMatrix;
int j = myMatrix.RowStart();
int i;
#pragma omp parallel
{
    #pragma omp for firstprivate(j) lastprivate(i)
    reduction(+: sum)
    for(i = 0; i < count; ++i)
    {
        int doubleI = 2 * i;
        for(; j < doubleI; ++j)
        {
            sum += myMatrix.GetElement(i, j);
        }
    }
}
```

Во-вторых, частными являются локальные переменные блоков параллельных регионов. На рис. 3 такова переменная `doubleI`, потому что она объявлена в параллельном регионе. Любые нестатические и не являющиеся членами класса `MatrixClass` переменные, объявленные в методе `myMatrix::GetElement`, будут частными.

В-третьих, частными будут любые переменные, указанные в разделах `private`, `firstprivate`, `lastprivate` и `reduction`. В листинге 2 переменные `i`, `j` и `sum` сделаны частными для каждого потока из группы, т. е. каждый поток будет располагать своей копией каждой из этих переменных.

Каждый из четырех названных разделов принимает список переменных, но семантика этих разделов различается. Раздел `private` говорит о том, что для каждого потока должна быть создана частная копия каждой переменной из списка. Частные копии будут инициализироваться значением по умолчанию (с применением конструктора по умолчанию, если это уместно). Например, переменные типа `int` имеют по умолчанию значение 0.

У раздела `firstprivate` такая же семантика, но перед выполнением параллельного региона он указывает копировать значение частной переменной в каждый поток, используя конструктор копий, если это уместно.

Семантика раздела `lastprivate` тоже совпадает с семантикой раздела `private`, но при выполнении последней итерации цикла или раздела конструкции распараллеливания значения переменных, указанных в разделе `lastprivate`, присваиваются переменным

основного потока. Если это уместно, для копирования объектов применяется оператор присваивания копий (copy assignment operator).

Похожая семантика и у раздела reduction, но он принимает переменную и оператор. Поддерживаемые этим разделом операторы перечислены в табл. 1, а у переменной должен быть скалярный тип (например, float, int или long, но не std::vector, int [] и т. д.). Переменная раздела reduction инициализируется в каждом потоке значением, указанным в таблице. В конце блока кода оператор раздела reduction применяется к каждой частной копии переменной, а также к исходному значению переменной.

Табл. 1. Операторы раздела reduction Оператор раздела reduction Инициализированное (каноническое) значение

+	0
*	1
-	0
&	~0 (каждый бит установлен)
	0
^	0
&&	1
	0

В листинге 2 переменная sum неявно инициализируется в каждом потоке значением 0.0f (заметьте, что в таблице указано каноническое значение 0, но в данном случае оно принимает форму 0.0f, так как sum имеет тип float). После выполнения блока #pragma omp for над всеми частными значениями и исходным значением sum (которое в нашем случае равно 10.0f) выполняется операция +. Результат присваивается исходной общей переменной sum.

Параллельная обработка в конструкциях, отличных от циклов

Как правило, OpenMP используется для распараллеливания циклов, но OpenMP поддерживает параллелизм и на уровне функций. Этот механизм называется секциями OpenMP (OpenMP sections). Он довольно прост и часто бывает полезен.

Рассмотрим один из самых важных алгоритмов в программировании — быструю сортировку (quicksort). В качестве примера мы реализовали рекурсивный метод быстрой сортировки списка целых чисел. Ради простоты мы решили не создавать универсальную шаблонную версию метода, но суть дела от этого ничуть не меняется. Код нашего метода, реализованного с использованием секций OpenMP, показан в листинге 3 (код метода Partition опущен, чтобы не загромождать общую картину).

Листинг 3. Быстрая сортировка с использованием параллельных секций

```
void QuickSort (int numList[], int nLower, int nUpper)
{
    if (nLower < nUpper)
    {
        // Разбиение интервала сортировки
        int nSplit = Partition (numList, nLower, nUpper);
        #pragma omp parallel sections
        {
```

```

#pragma omp section
QuickSort (numList, nLower, nSplit - 1);

#pragma omp section
QuickSort (numList, nSplit + 1, nUpper);
}
}
}

```

В данном примере первая директива `#pragma` создает параллельный регион секций. Каждая секция определяется директивой `#pragma omp section`. Каждой секции в параллельном регионе ставится в соответствие один поток из группы потоков, и все секции выполняются одновременно. В каждой секции рекурсивно вызывается метод `QuickSort`.

Как и в случае конструкции `#pragma omp parallel for`, вы сами должны убедиться в независимости секций друг от друга, чтобы они могли выполняться параллельно. Если в секциях изменяются общие ресурсы без синхронизации доступа к ним, результат может оказаться непредсказуемым.

Обратите внимание на то, что в этом примере используется сокращение `#pragma omp parallel sections`, аналогичное конструкции `#pragma omp parallel for`. По аналогии с `#pragma omp for` директиву `#pragma omp sections` можно использовать в параллельном регионе отдельно.

По поводу кода, показанного в листинге 3, следует сказать еще пару слов. Прежде всего заметьте, что параллельные секции вызываются рекурсивно. Рекурсивные вызовы поддерживаются и параллельными регионами, и (как в нашем примере) параллельными секциями. Если создание вложенных секций разрешено, по мере рекурсивных вызовов `QuickSort` будут создаваться все новые и новые потоки. Возможно, это не то, что нужно программисту, так как такой подход может привести к созданию большого числа потоков. Чтобы ограничить число потоков, в программе можно запретить вложение. Тогда наше приложение будет рекурсивно вызывать метод `QuickSort`, используя только два потока.

При компиляции этого приложения без параметра `/openmp` будет сгенерирована корректная последовательная версия. Одно из преимуществ OpenMP в том, что эта технология совместима с компиляторами, не поддерживающими OpenMP.

Директивы `pragma` для синхронизации

При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. OpenMP поддерживает несколько типов синхронизации, помогающих во многих ситуациях.

Один из типов — неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.

Неявная барьерная синхронизация выполняется также в конце каждого блока `#pragma omp for`, `#pragma omp single` и `#pragma omp sections`. Чтобы отключить неявную барьерную

синхронизацию в каком-либо из этих трех блоков разделения работы, укажите раздел `nowait`:

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(int i = 1; i < size; ++i)
        x[i] = (y[i-1] + y[i+1])/2;
}
```

Как видите, этот раздел директивы распараллеливания говорит о том, что синхронизировать потоки в конце цикла `for` не надо, хотя в конце параллельного региона они все же будут синхронизированы.

Второй тип — явная барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Для этого включите в код директиву `#pragma omp barrier`.

В качестве барьеров можно использовать критические секции. В Win32 API для входа в критическую секцию и выхода из нее служат функции `EnterCriticalSection` и `LeaveCriticalSection`. В OpenMP для этого применяется директива `#pragma omp critical [имя]`. Она имеет такую же семантику, что и критическая секция Win32, и опирается на `EnterCriticalSection`. Вы можете использовать именованную критическую секцию, и тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой. Доступ ко всем именованным критическим секциям является взаимоисключающим.

В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, — например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в OpenMP служит директива `#pragma omp single`.

Иногда возможностей директивы `single` недостаточно. В ряде случаев требуется, чтобы блок кода был выполнен основным потоком, — например, если этот поток отвечает за обработку GUI и вам нужно, чтобы какую-то задачу выполнил именно он. Тогда применяется директива `#pragma omp master`. В отличие от директивы `single` при входе в блок `master` и выходе из него нет никакого неявного барьера.

Чтобы завершить все незавершенные операции над памятью перед началом следующей операции, используйте директиву `#pragma omp flush`, которая эквивалентна внутренней функции компилятора `_ReadWriteBarrier`.

Учтите, что OpenMP-директивы `pragma` должны обрабатываться всеми потоками из группы в одном порядке (или вообще не обрабатываться никакими потоками). Таким образом, следующий пример кода некорректен, а предсказать результаты его выполнения нельзя (вероятные варианты — сбой или зависание системы):

```
#pragma omp parallel
{
    if(omp_get_thread_num() > 3)
    {
        #pragma omp single // код, доступный не всем потокам
        x++;
    }
}
```

```
}  
}
```

Подпрограммы исполняющей среды OpenMP

Помимо уже описанных директив OpenMP поддерживает ряд полезных подпрограмм. Они делятся на три обширных категории: функции исполняющей среды, блокировки/синхронизации и работы с таймерами (последние в этой статье не рассматриваются). Все эти функции имеют имена, начинающиеся с `omp_`, и определены в заголовочном файле `omp.h`.

Подпрограммы первой категории позволяют запрашивать и задавать различные параметры операционной среды OpenMP. Функции, имена которых начинаются на `omp_set_`, можно вызывать только вне параллельных регионов. Все остальные функции можно использовать как внутри параллельных регионов, так и вне таковых.

Чтобы узнать или задать число потоков в группе, используйте функции `omp_get_num_threads` и `omp_set_num_threads`. Первая возвращает число потоков, входящих в текущую группу потоков. Если вызывающий поток выполняется не в параллельном регионе, эта функция возвращает 1. Метод `omp_set_num_threads` задает число потоков для выполнения следующего параллельного региона, который встретится текущему выполняемому потоку. Кроме того, число потоков, используемых для выполнения параллельных регионов, зависит от двух других параметров среды OpenMP: поддержки динамического создания потоков и вложения регионов.

Поддержка динамического создания потоков определяется значением булевого свойства, которое по умолчанию равно `false`. Если при входе потока в параллельный регион это свойство имеет значение `false`, исполняющая среда OpenMP создает группу, число потоков в которой равно значению, возвращаемому функцией `omp_get_max_threads`. По умолчанию `omp_get_max_threads` возвращает число потоков, поддерживаемых аппаратно, или значение переменной `OMP_NUM_THREADS`. Если поддержка динамического создания потоков включена, исполняющая среда OpenMP создаст группу, которая может содержать переменное число потоков, не превышающее значение, которое возвращается функцией `omp_get_max_threads`.

Вложение параллельных регионов также определяется булевым свойством, которое по умолчанию установлено в `false`. Вложение параллельных регионов происходит, когда потоку, уже выполняющему параллельный регион, встречается другой параллельный регион. Если вложение разрешено, создается новая группа потоков, при этом соблюдаются правила, описанные ранее. А если вложение не разрешено, формируется группа, содержащая один поток.

Для установки и чтения свойств, определяющих возможность динамического создания потоков и вложения параллельных регионов, служат функции `omp_set_dynamic`, `omp_get_dynamic`, `omp_set_nested` и `omp_get_nested`. Кроме того, каждый поток может запросить информацию о своей среде. Чтобы узнать номер потока в группе потоков, вызовите `omp_get_thread_num`. Помните, что она возвращает не Windows-идентификатор потока, а число в диапазоне от 0 до `omp_get_num_threads - 1`.

Функция `omp_in_parallel` позволяет потоку узнать, выполняет ли он в настоящее время параллельный регион, а `omp_get_num_procs` возвращает число процессоров в компьютере.

Чтобы лучше понять взаимосвязи различных функций исполняющей среды, взгляните на листинг 4. В этом примере мы реализовали четыре отдельных параллельных региона и два вложенных.

Листинг 4. Использование подпрограмм исполняющей среды OpenMP

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

int main()
{
    omp_set_dynamic(1);
    omp_set_num_threads(10);
    #pragma omp parallel // параллельный регион 1
    {
        #pragma omp single
        printf("Num threads in dynamic region is = %d\n",
            omp_get_num_threads());
    }
    printf("\n");
    omp_set_dynamic(0);
    omp_set_num_threads(10);
    #pragma omp parallel // параллельный регион 2
    {
        #pragma omp single
        printf("Num threads in non-dynamic region is = %d\n",
            omp_get_num_threads());
    }
    printf("\n");
    omp_set_dynamic(1);
    omp_set_num_threads(10);
    #pragma omp parallel // параллельный регион 3
    {
        #pragma omp parallel
        {
            #pragma omp single
            printf(
                "Num threads in nesting disabled region is = %d\n",
                omp_get_num_threads());
        }
    }
    printf("\n");
    omp_set_nested(1);
    #pragma omp parallel // параллельный регион 4
    {
        #pragma omp parallel
        {
            #pragma omp single
            printf("Num threads in nested region is = %d\n",
                omp_get_num_threads());
        }
    }
}
```

```
}
```

Скомпилировав этот код в Visual Studio 2005 и выполнив его на обычном двухпроцессорном компьютере, мы получили такой результат:

```
Num threads in dynamic region is = 2
```

```
Num threads in non-dynamic region is = 10
```

```
Num threads in nesting disabled region is = 1
```

```
Num threads in nesting disabled region is = 1
```

```
Num threads in nested region is = 2
```

```
Num threads in nested region is = 2
```

Для первого региона мы включили динамическое создание потоков и установили число потоков в 10. По результатам работы программы видно, что при включенном динамическом создании потоков исполняющая среда OpenMP решила создать группу, включающую всего два потока, так как у компьютера два процессора. Для второго параллельного региона исполняющая среда OpenMP создала группу из 10 потоков, потому что динамическое создание потоков для этого региона было отключено.

Результаты выполнения третьего и четвертого параллельных регионов иллюстрируют следствия включения и отключения возможности вложения регионов. В третьем параллельном регионе вложение было отключено, поэтому для вложенного параллельного региона не было создано никаких новых потоков — и внешний, и вложенный параллельные регионы выполнялись двумя потоками. В четвертом параллельном регионе, где вложение было включено, для вложенного параллельного региона была создана группа из двух потоков (т. е. в общей сложности этот регион выполнялся четырьмя потоками). Процесс удвоения числа потоков для каждого вложенного параллельного региона может продолжаться, пока вы не исчерпаете пространство в стеке. На практике можно создать несколько сотен потоков, хотя связанные с этим издержки легко перевесят любые преимущества.

Как вы, вероятно, заметили, для третьего и четвертого параллельных регионов динамическое создание потоков было включено. Посмотрим, что будет, если выполнить тот же код, отключив динамическое создание потоков:

```
omp_set_dynamic(0);
omp_set_nested(1);
omp_set_num_threads(10);
#pragma omp parallel
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Num threads in nested region is = %d\n",
            omp_get_num_threads());
    }
}
```

А происходит то, чего и следовало ожидать. Для первого параллельного региона создается группа из 10 потоков, затем при входе во вложенный параллельный регион для каждого из

этих 10 потоков создается группа также из 10 потоков. В общей сложности вложенный параллельный регион выполняют 100 потоков:

```
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10  
Num threads in nested region is = 10
```

Методы синхронизации/блокировки

OpenMP включает и функции, предназначенные для синхронизации кода. В OpenMP два типа блокировок: простые и вкладываемые (nestable); блокировки обоих типов могут находиться в одном из трех состояний — неинициализированном, заблокированном и разблокированном.

Простые блокировки (omp_lock_t) не могут быть установлены более одного раза, даже тем же потоком. Вкладываемые блокировки (omp_nest_lock_t) идентичны простым с тем исключением, что, когда поток пытается установить уже принадлежащую ему вкладываемую блокировку, он не блокируется. Кроме того, OpenMP ведет учет ссылок на вкладываемые блокировки и следит за тем, сколько раз они были установлены.

OpenMP предоставляет подпрограммы, выполняющие операции над этими блокировками. Каждая такая функция имеет два варианта: для простых и для вкладываемых блокировок. Вы можете выполнить над блокировкой пять действий: инициализировать ее, установить (захватить), освободить, проверить и уничтожить. Все эти операции очень похожи на Win32-функции для работы с критическими секциями, и это не случайность: на самом деле технология OpenMP реализована как оболочка этих функций. Соответствие между функциями OpenMP и Win32 иллюстрирует табл. 2.

OpenMP	Вложенная блокировка OpenMP	Win32-функция
omp_lock_t	omp_nest_lock_t	CRITICAL_SECTION
omp_init_lock	omp_init_nest_lock	InitializeCriticalSection
omp_destroy_lock	omp_destroy_nest_lock	DeleteCriticalSection
omp_set_lock	omp_set_nest_lock	EnterCriticalSection
omp_unset_lock	omp_unset_nest_lock	LeaveCriticalSection
omp_test_lock	omp_test_nest_lock	TryEnterCriticalSection

Для синхронизации кода можно использовать и подпрограммы исполняющей среды, и директивы синхронизации. Преимущество директив в том, что они прекрасно структурированы. Это делает их более понятными и облегчает поиск мест входа в синхронизированные регионы и выхода из них.

Преимущество подпрограмм исполняющей среды — гибкость. Например, вы можете передать блокировку в другую функцию и установить/освободить ее в этой функции. При использовании директив это невозможно. Как правило, если вам не нужна гибкость,

обеспечиваемая лишь подпрограммами исполняющей среды, лучше использовать директивы синхронизации.

Параллельная обработка структур данных

В листинге 5 показан код двух параллельно выполняемых циклов, в начале которых исполняющей среде неизвестно число их итераций. В первом примере выполняется перебор элементов STL-контейнера `std::vector`, а во втором — стандартного связанного списка.

Листинг 5. Выполнение заранее неизвестного числа итераций

```
#pragma omp parallel
{
    // Параллельная обработка вектора STL
    std::vector<int>::iterator iter;
    for(iter = xVect.begin(); iter != xVect.end(); ++iter)
    {
        #pragma omp single nowait
        {
            process1(*iter);
        }
    }

    // Параллельная обработка стандартного связанного списка
    for(LList *listWalk = listHead; listWalk != NULL;
        listWalk = listWalk->next)
    {
        #pragma omp single nowait
        {
            process2(listWalk);
        }
    }
}
```

В примере с вектором STL каждый поток из группы потоков выполняет цикл `for` и имеет собственный экземпляр итератора, но при каждой итерации лишь один поток входит в блок `single` (такова семантика директивы `single`). Все действия, гарантирующие однократное выполнение блока `single` при каждой итерации, берет на себя исполняющая среда OpenMP. Такой способ выполнения цикла сопряжен со значительными издержками, поэтому он полезен, только если в функции `process1` выполняется много работы. В примере со связанным списком реализована та же логика.

Стоит отметить, что в примере с вектором STL мы можем до входа в цикл определить число его итераций по значению `std::vector.size`, что позволяет привести цикл к канонической форме для OpenMP:

```
#pragma omp parallel for
for(int i = 0; i < xVect.size(); ++i)
    process(xVect[i]);
```

Это существенно уменьшает издержки в период выполнения, и именно такой подход мы рекомендуем применять для обработки массивов, векторов и любых других контейнеров, элементы которых можно перебрать в цикле `for`, соответствующем канонической форме для OpenMP.

Более сложные алгоритмы планирования

По умолчанию в OpenMP для планирования параллельного выполнения циклов `for` применяется алгоритм, называемый статическим планированием (static scheduling). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если n — число итераций цикла, а T — число потоков в группе, каждый поток выполнит n/T итераций (если n не делится на T без остатка, ничего страшного). Однако OpenMP поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (dynamic scheduling), планирование в период выполнения (runtime scheduling) и управляемое планирование (guided scheduling).

Чтобы задать один из этих механизмов планирования, используйте раздел `schedule` в директиве `#pragma omp for` или `#pragma omp parallel for`. Формат этого раздела выглядит так:

`schedule(алгоритм планирования[, число итераций])`

Вот примеры этих директив:

```
#pragma omp parallel for schedule(dynamic, 15)
for(int i = 0; i < 100; ++i)
...
#pragma omp parallel
  #pragma omp for schedule(guided)
```

При динамическом планировании каждый поток выполняет указанное число итераций. Если это число не задано, по умолчанию оно равно 1. После того как поток завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации. Последний набор итераций может быть меньше, чем изначально заданный.

При управляемом планировании число итераций, выполняемых каждым потоком, определяется по следующей формуле:

число_выполняемых_потоком_итераций =
 $\max(\text{число_нераспределенных_итераций}/\text{omp_get_num_threads}(),$
число итераций)

Завершив выполнение назначенных итераций, поток запрашивает выполнение другого набора итераций, число которых определяется по только что приведенной формуле. Таким образом, число итераций, назначаемых каждому потоку, со временем уменьшается. Последний набор итераций может быть меньше, чем значение, вычисленное по формуле.

Если указать директиву `#pragma omp for schedule(dynamic, 15)`, цикл `for` из 100 итераций может быть выполнен четырьмя потоками следующим образом:

```
Поток 0 получает право на выполнение итераций 1-15
Поток 1 получает право на выполнение итераций 16-30
Поток 2 получает право на выполнение итераций 31-45
Поток 3 получает право на выполнение итераций 46-60
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 61-75
Поток 3 завершает выполнение итераций
Поток 3 получает право на выполнение итераций 76-90
Поток 0 завершает выполнение итераций
Поток 0 получает право на выполнение итераций 91-100
```

А вот каким может оказаться результат выполнения того же цикла четырьмя потоками, если будет указана директива `#pragma omp for schedule(guided, 15)`:

Поток 0 получает право на выполнение итераций 1-25
Поток 1 получает право на выполнение итераций 26-44
Поток 2 получает право на выполнение итераций 45-59
Поток 3 получает право на выполнение итераций 60-64
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 65-79
Поток 3 завершает выполнение итераций
Поток 3 получает право на выполнение итераций 80-94
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 95-100

Динамическое и управляемое планирование хорошо подходят, если при каждой итерации выполняются разные объемы работы или если одни процессоры более производительны, чем другие. При статическом планировании нет никакого способа, позволяющего сбалансировать нагрузку на разные потоки. При динамическом и управляемом планировании нагрузка распределяется автоматически — такова сама суть этих подходов. Как правило, при управляемом планировании код выполняется быстрее, чем при динамическом, вследствие меньших издержек на планирование.

Последний подход — планирование в период выполнения — это скорее даже не алгоритм планирования, а способ динамического выбора одного из трех описанных алгоритмов. Если в разделе `schedule` указан параметр `runtime`, исполняющая среда OpenMP использует алгоритм планирования, заданный для конкретного цикла `for` при помощи переменной `OMP_SCHEDULE`. Она имеет формат «тип[,число итераций]», например:
`set OMP_SCHEDULE=dynamic,8`

Планирование в период выполнения дает определенную гибкость в выборе типа планирования, при этом по умолчанию применяется статическое планирование.

Когда использовать OpenMP?

Знать, когда использовать технологию OpenMP, не менее важно, чем уметь с ней работать. Надеемся, что наши советы вам помогут.

Целевая платформа является многопроцессорной или многоядерной. Если приложение полностью использует ресурсы одного ядра или процессора, то, сделав его многопоточным при помощи OpenMP, вы почти наверняка повысите его быстродействие.

Приложение должно быть кроссплатформенным. OpenMP — кроссплатформенный и широко поддерживаемый API. А так как он реализован на основе директив `pragma`, приложение можно скомпилировать даже при помощи компилятора, не поддерживающего стандарт OpenMP.

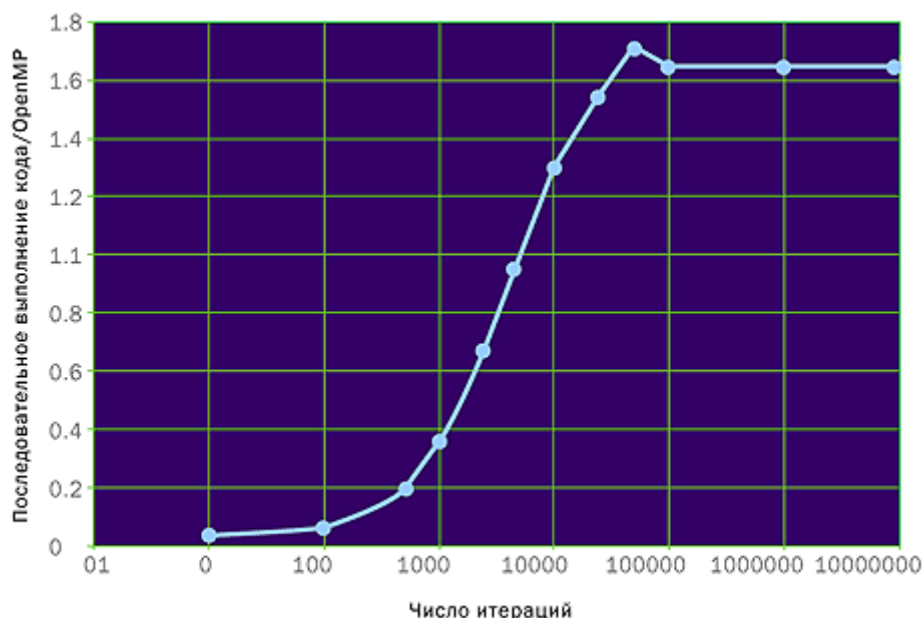
Выполнение циклов нужно распараллелить. Весь свой потенциал OpenMP демонстрирует при организации параллельного выполнения циклов. Если в приложении есть длительные циклы без зависимостей, OpenMP — идеальное решение.

Перед выпуском приложения нужно повысить его быстродействие. Так как технология OpenMP не требует переработки архитектуры приложения, она прекрасно подходит для внесения в код небольших изменений, позволяющих повысить его быстродействие.

В то же время следует признать, что OpenMP — не панацея от всех бед. Эта технология ориентирована в первую очередь на разработчиков высокопроизводительных вычислительных систем и наиболее эффективна, если код включает много циклов и работает с разделяемыми массивами данных.

Создание как обычных потоков, так и параллельных регионов OpenMP имеет свою цену. Чтобы применение OpenMP стало выгодным, выигрыш в скорости, обеспечиваемый параллельным регионом, должен превосходить издержки на создание группы потоков. В версии OpenMP, реализованной в Visual C++, группа потоков создается при входе в первый параллельный регион. После завершения региона группа потоков приостанавливается, пока не понадобится вновь. За кулисами OpenMP использует пул потоков Windows. Рис. 2 иллюстрирует прирост быстродействия простой программы, приведенной в начале статьи, который достигается благодаря OpenMP на двухпроцессорном компьютере при различном числе итераций. Максимальный прирост быстродействия составляет примерно 1,7 от исходного, что типично для двухпроцессорных систем.

Рис. 2. Сравнение времени последовательного и параллельного выполнения кода в двухпроцессорной системе



На данном графике ось y представляет отношение времени последовательного выполнения кода ко времени параллельного выполнения того же кода. Обратите внимание, что параллельная версия настигает по быстродействию последовательную примерно при 5000 итераций, но это почти худший сценарий. Большинство параллельных циклов будут выполняться быстрее последовательных даже при значительно меньшем числе итераций. Это зависит от объема работы, выполняемой на каждой итерации. Как бы то ни было, этот график показывает, насколько важно оценивать производительность ПО. Само по себе применение OpenMP не гарантирует, что быстродействие вашего кода повысится.

OpenMP-директивы `pragma` просты в использовании, но не позволяют получать детальные сведения об ошибках. Если вы пишете критически важное приложение, которое должно определять ошибки и корректно восстанавливать нормальную работу, от OpenMP, пожалуй, следует отказаться (по крайней мере, пока). Например, если OpenMP не может создать потоки для параллельных регионов или критическую секцию, поведение программы становится неопределенным. В Visual C++ 2005 исполняющая среда OpenMP какое-то время продолжает пытаться выполнить нужную задачу, после чего сдается. В

будущих версиях OpenMP мы помимо прочего собираемся реализовать стандартный механизм уведомления об ошибках.

Еще одна ситуация, в которой следует сохранять бдительность, имеет место при использовании потоков Windows вместе с потоками OpenMP. Потоки OpenMP создаются на основе потоков Windows, поэтому они прекрасно работают в одном процессе. Увы, OpenMP ничего не знает о потоках Windows, созданных другими модулями. Из этого вытекают две проблемы: во-первых, исполняющая среда OpenMP не ведет учет других потоков Windows, а во-вторых, методы синхронизации OpenMP не синхронизируют потоки Windows, потому что они не входят в группы потоков.

Ловушки, в которые можно попасть при использовании OpenMP

Хотя использовать OpenMP совсем несложно, некоторые моменты все же требуют к себе повышенного внимания. Например, индексная переменная самого внешнего параллельного цикла `for` является частной, а индексные переменные вложенных циклов `for` по умолчанию общие. При работе с вложенными циклами обычно требуется, чтобы индексы внутренних циклов были частными. Используйте для этого раздел `private`.

Разрабатывая приложения OpenMP, следует быть осторожным при генерации исключений C++. Если приложение генерирует исключение в параллельном регионе, оно должно быть обработано в том же регионе тем же потоком. Иначе говоря, исключение не должно покинуть регион. Как правило, все исключения, которые могут быть сгенерированы в параллельном регионе, следует перехватывать. Если не перехватить исключение в том же параллельном регионе, приложение скорее всего потерпит крах.

Чтобы можно было открыть структурированный блок, выражение

```
#pragma omp <директива> [раздел]
```

должно завершаться символом новой строки, а не фигурной скобкой. Директива, заканчивающаяся фигурной скобкой, приведет к ошибке компиляции1:

```
// Плохо
```

```
#pragma omp parallel
```

```
{
```

```
// Ошибка компиляции
```

```
}
```

```
// Хорошо
```

```
#pragma omp parallel
```

```
{
```

```
    // Код
```

```
}
```

Отлаживать приложения OpenMP в среде Visual Studio 2005 иногда трудно. В частности, определенные неудобства связаны со входом в параллельный регион и/или с выходом из него нажатием клавиши F10/F11. Это объясняется тем, что компилятор генерирует дополнительный код для вызова исполняющей среды и групп потоков. Отладчик об этом не знает, поэтому то, что вы увидите, может показаться вам странным. Мы рекомендуем установить точку прерывания в параллельном регионе и нажать F5, чтобы достичь ее. Чтобы выйти из параллельного региона, установите точку прерывания вне такого региона и нажмите F5.

При нахождении внутри параллельного региона в окне Threads Window отладчика будет отображаться информация о потоках, выполняемых в группе потоков. Идентификаторы этих потоков будут соответствовать не потокам OpenMP, а лежащим в их основе потокам Windows.

В настоящее время использовать с OpenMP оптимизацию, определяемую профилем (Profile Guided Optimization, PGO), нельзя. К счастью, технология OpenMP основана на директивах pragma, поэтому вы можете скомпилировать свое приложение с параметром /openmp и с PGO и узнать, какой подход более эффективен.

OpenMP и .NET

Высокопроизводительные вычисления мало у кого ассоциируются с .NET, но в Visual C++ 2005 эта ситуация улучшена. Особо стоит отметить то, что мы добились совместной работы OpenMP с управляемым C++-кодом. Для этого мы обеспечили совместимость параметра /openmp с /clr и /clr:OldSyntax. То есть вы можете использовать OpenMP для параллельного выполнения методов .NET-типов, которые подлежат сбору мусора. Учтите, что сейчас параметр /openmp не совместим ни с /clr:safe, ни с /clr:pure, но мы планируем исправить это.

Мы должны упомянуть одно важное ограничение, связанное с применением OpenMP в управляемом коде. Приложение, в котором задействованы средства OpenMP, следует использовать только в одном домене приложения. При загрузке другого AppDomain в процесс с уже загруженной исполняющей средой OpenMP приложение может потерпеть крах.

OpenMP — простая, но мощная технология распараллеливания приложений. Она позволяет реализовать параллельное выполнение как циклов, так и функциональных блоков кода. Она легко интегрируется в существующие приложения и включается/выключается одним параметром компилятора. OpenMP позволяет более полно использовать вычислительную мощь многоядерных процессоров. Мы настоятельно советуем вам ознакомиться со спецификацией OpenMP. Удачи в разработке многопоточных программ!

1 Пример, приведенный авторами ниже, противоречит этому утверждению. Здесь авторами, видимо, допущена какая-то ошибка. — Прим. перев.