Аннотация

С распространением многоядерных систем задача параллельного программирования становится все более и более актуальной. Данная область, однако, является новой даже для большинства опытных программистов. Существующие компиляторы и анализаторы кода позволяют находить некоторые ошибки, возникающие при разработке параллельного кода. Многие ошибки никак не диагностируются. В данной статье приводится описание ряда ошибок, приводящих к некорректному поведению параллельных программ, созданных на основе технологии OpenMP.

Введение

Параллельное программирование появилось уже достаточно давно. Первый многопроцессорный компьютер был создан еще в 60-х годах прошлого века. Однако до недавних пор прирост производительности процессоров обеспечивался в основном благодаря росту тактовой частоты, и многопроцессорные системы были редкостью. Сейчас рост тактовой частоты замедляется, и прирост производительности обеспечивается за счет использования нескольких ядер. Многоядерные процессоры приобретают широкое распространение, и в связи с этим задача написания параллельных программ приобретает все большую и большую актуальность. Если ранее для увеличения производительности программы пользователь мог просто установить процессор с большей тактовой частотой и большим кешем, то теперь такой подход невозможен и существенное увеличение производительности в любом случае потребует усилий от программиста.

В связи с тем, что параллельное программирование начинает набирать популярность только сейчас, процесс распараллеливания существующего приложения или написания нового параллельного кода может вызвать проблемы даже для опытных программистов, так как данная область является для них новой. Существующие компиляторы и анализаторы кода позволяют диагностировать лишь некоторые из возможных ошибок. Остальные ошибки (а таких большинство) остаются неучтенными и могут существенно увеличить время тестирования и отладки, особенно с учетом того, что такие ошибки почти всегда воспроизводятся нестабильно. В данной статье рассматривается язык Си++, поскольку к коду именно на этом языке чаще всего предъявляются требования высокой производительности. Так как поддержка технологии OpenMP встроена в Microsoft Visual Studio 2005 и 2008 (заявлено соответствие стандарту OpenMP 2.0), мы будем рассматривать именно эту технологию. OpenMP позволяет с минимальными затратами переделать существующий код - достаточно лишь включить дополнительный флаг компилятора /openmp и добавить в свой код соответствующие директивы, описывающие, как именно выполнение программы будет распределено на несколько процессоров.

В данной статье рассматриваются далеко не все ошибки, которые могут возникнуть при программировании на Си++ с использованием OpenMP и не диагностируются при этом компилятором или существующими статическими или динамическими анализаторами кода. Тем не менее, хочется надеяться, что приведенный материал поможет лучше понять особенности разработки параллельных систем и избежать множества ошибок.

Также стоит отметить, что данная статья носит исследовательский характер, и ее материал послужит при разработке статического анализатора VivaMP (http://www.viva64.com/vivamp.php), предназначенного для поиска ошибок в параллельных программах, создаваемых на основе технологии OpenMP. Мы будем рады получить отзыв о статье и узнать новые паттерны параллельных ошибок.

По аналогии с одной из использованных статей [1] ошибки в данной статье разделены на логические ошибки и ошибки производительности. Логические ошибки это ошибки, приводящие к неожиданным результатам, то есть к некорректной работе программы. Под ошибками производительности понимаются ошибки, приводящие к снижению быстродействия программы.

Прежде чем перейти к рассмотрению конкретных ошибок, определим некоторые специфические термины, которые будут использоваться в этой статье:

Директивами (directives) назовем собственно директивы OpenMP, определяющие способ распараллеливания кода. Все директивы OpenMP имеют вид #pragma omp ...

Выражением (clause) будем называть вспомогательные части директив, определяющие количество потоков, режим распределения работы между потоками, режим доступа к переменным, и т. п.

Параллельной секцией (section) назовем фрагмент кода, на который распространяется действие директивы #pragma omp parallel.

Данная статья предполагает, что читатель уже знаком с основами OpenMP и собирается применять эту технологию в своих программах. Если же читатель еще не знаком с OpenMP, для первоначального знакомства можно посмотреть документ [2]. Более подробное описание директив, выражений, функций и глобальных переменных OpenMP можно найти в спецификации OpenMP 2.0 [3]. Также эта спецификация продублирована в справочной системе MSDN Library, в более удобной форме, чем формат PDF.

Теперь перейдем к рассмотрению возможных ошибок, не диагностируемых совсем или плохо диагностируемых стандартными компиляторами.

Логические ошибки

1. Отсутствие /openmp

Начнем с самой простейшей ошибки: если поддержка OpenMP не будет включена в настройках компилятора, директивы OpenMP будут попросту игнорироваться. Компилятор не выдаст ни ошибки, ни даже предупреждения, код просто будет выполняться не так, как этого ожидает программист.

Поддержку OpenMP можно включить в диалоговом окне свойств проекта (раздел "Configuration Properties | C/C++ | Language").

2. Отсутствие parallel

Директивы OpenMP имеют достаточно сложный формат, поэтому сначала рассмотрим простейшие ошибки, которые могут возникнуть при неправильном написании самих директив. Ниже приведены примеры корректного и некорректного кода:

Некорректно:

```
#pragma omp for
... //код
```

Корректно:

```
#pragma omp parallel for
... // код
#pragma omp parallel
{
  #pragma omp for
  ... //код
}
```

Первый фрагмент кода успешно скомпилируется, и директива #pragma omp for будет попросту проигнорирована компилятором. Таким образом, цикл, вопреки ожиданиям программиста, будет выполняться только одним потоком, и обнаружить это будет достаточно затруднительно. Помимо директивы #pragma omp parallel for, эта ошибка может возникнуть и с директивой #pragma omp parallel sections.

3. Отсутствие omp

Ситуация, аналогичная предыдущей, возникнет, если в директиве OpenMP не написать ключевое слово omp [4]. Рассмотрим простой пример.

Некорректно:

```
#pragma omp parallel num_threads(2)
{
   #pragma single
   {
     printf("me\n");
   }
}
```

Корректно:

```
#pragma omp parallel num_threads(2)
{
   #pragma omp single
   {
     printf("me\n");
   }
}
```

При выполнении этого кода сообщение "me" будет выведено два раза вместо ожидаемого одного. При компиляции компилятор выдаст предупреждение: "warning C4068: unknown pragma". Однако предупреждения могут быть отключены в настройках проекта, либо предупреждение может быть проигнорировано программистом.

4. Отсутствие for

Директива #pragma omp parallel может относиться как к блоку кода, так и к одной команде. В случае цикла for это может приводить к неожиданному поведению:

```
#pragma omp parallel num_threads(2)
for (int i = 0; i < 10; i++)
    myFunc();
```

Если программист хотел распределить выполнение этого цикла на два потока, ему следовало использовать директиву #pragma omp parallel for. В этом случае цикл выполнился бы 10 раз. Однако, при запуске приведенного выше кода цикл будет выполняться по одному разу в каждом потоке, и вместо ожидаемых 10 раз функция myFunc будет вызвана 20 раз. Исправленная версия кода должна выглядеть следующим образом:

```
#pragma omp parallel for num_threads(2)
for (int i = 0; i < 10; i++)
    myFunc();
```

5. Ненужное распараллеливание

Применение директивы #pragma omp parallel к большому участку кода при невнимательности программиста может вызывать неожиданное поведение, аналогичное предыдущему случаю:

```
#pragma omp parallel num_threads(2)
{
    ... // N строк кода
    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
    {
           myFunc();
    }
}
```

В силу забывчивости или неопытности программист, желающий распределить выполнение цикла на два потока, написал слово parallel внутри секции кода, уже распределенного на два потока. В результате выполнения этого кода функция myFunc, как и в предыдущем примере, будет выполнена 20 раз, вместо ожидаемых 10 раз. Исправленный код должен выглядеть так:

```
#pragma omp parallel num_threads(2)
{
    ... // N строк кода
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
          myFunc();
    }
}
```

6. Неправильное применение ordered

Применение директивы ordered может вызвать проблемы у начинающих программистов, не слишком хорошо знакомых с OpenMP [1]. Приведем пример кода:

Некорректно:

```
#pragma omp parallel for ordered
for (int i = 0; i < 10; i++)
{
    myFunc(i);
}
```

Корректно:

```
#pragma omp parallel for ordered
for (int i = 0; i < 10; i++)
{
    #pragma omp ordered
    {
          myFunc(i);
    }
}
```

Суть ошибки заключается в том, что в первом примере выражение ordered будет просто проигнорировано, поскольку не указана область его действия. Цикл будет выполняться в произвольном порядке.

7. Переопределение количества потоков внутри параллельной секции

Перейдем к рассмотрению более сложных ошибок, которые могут возникнуть при недостаточном знании стандарта OpenMP. Согласно спецификации OpenMP 2.0 [3], количество потоков нельзя переопределять внутри параллельной секции. В Си++ это приводит к ошибкам во время выполнения программы и ее аварийному завершению. Пример:

Некорректно:

```
#pragma omp parallel
{
    omp_set_num_threads(2);
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
          myFunc();
    }
}
```

Корректно:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
           myFunc();
    }
}
```

Корректно:

```
omp_set_num_threads(2)
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
           myFunc();
    }
}
```

8. Попытка использовать блокировку без инциализации переменной

Согласно спецификации OpenMP 2.0 [3], переменные, использующиеся для блокировки, необходимо инициализировать перед использованием, вызвав функцию omp_init_lock или omp_init_nest_lock (в зависимости от типа переменной). В C++ попытка использования (установка блокировки, снятие блокировки, проверка блокировки) неинициализированной переменной приводит к ошибке во время выполнения программы.

Некорректно:

```
omp_lock_t myLock;
#pragma omp parallel num_threads(2)
{
    ...
    omp_set_lock(&myLock);
    ...
}
```

Корректно:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel num_threads(2)
{
    ...
    omp_set_lock(&myLock);
    ...
}
```

9. Попытка снять блокировку не из того потока, который ее установил

Если блокировка установлена одним потоком, попытка ее снятия из другого потока может привести к непредсказуемым результатам [3]. Рассмотрим пример:

Некорректно:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
    }
    #pragma omp section
    {
          ...
          omp_unset_lock(&myLock);
          ...
    }
}
```

В Си++ этот код приводит к ошибке во время выполнения программы. Поскольку операции установки и снятия блокировки по сути аналогичны входу в критическую секцию и выходу из нее, каждый из использующих блокировку потоков должен выполнять обе операции. Корректная версия кода будет выглядеть следующим образом:

Корректно:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
          omp_unset_lock(&myLock);
          ...
```

```
    }
    #pragma omp section
    {
            ...
            omp_set_lock(&myLock);
            ...
            omp_unset_lock(&myLock);
            ...
    }
}
```

10. Попытка использования блокировки как барьера

Функция omp_set_lock блокирует выполнение потока до тех пор, пока переменная, использующаяся для блокировки, не станет доступна, то есть до тех пор, пока тот же самый поток не вызовет функцию omp_unset_lock. Следовательно, как уже говорилось в описании предыдущей ошибки, каждый поток должен содержать вызовы обеих функций. Однако, начинающий программист, недостаточно хорошо понимающий принципы работы OpenMP, может попытаться использовать функцию omp_set_lock в качестве барьера, то есть вместо директивы #pragma omp barrier (эту директиву нельзя использовать внутри параллельных секций, к которым применяется директива #pragma omp sections). В результате возникнет следующий код.

Некорректно:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
            ...
            omp_set_lock(&myLock);
            ...
    }
    #pragma omp section
    {
            ...
            omp_set_lock(&myLock);
            omp_unset_lock(&myLock);
            ...
    }
}
```

В результате выполнения этого фрагмента кода программа иногда будет зависать, а иногда - выполняться нормально. Зависеть это будет от того, какой поток завершается последним. Если последним будет завершаться поток, в котором выполняется блокировка переменной без ее освобождения, программа будет выдавать ожидаемый результат. Во всех остальных случаях будет возникать бесконечное ожидание освобождения переменной, захваченной потоком, работающим с переменной некорректно. Аналогичная ситуация будет возникать и при использовании функции omp_test_lock в цикле (а именно так эту функцию обычно и используют). В этом случае поток будет бесконечно "топтаться на месте", так и не дождавшись освобождения переменной.

Поскольку данная ошибка аналогична предыдущей, корректный вариант кода будет выглядеть так же.

Корректно:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
            ...
            omp_set_lock(&myLock);
            ...
            omp_unset_lock(&myLock);
```

```
            ...
    }
    #pragma omp section
    {
            ...
            omp_set_lock(&myLock);
            ...
            omp_unset_lock(&myLock);
            ...
    }
}
```

11. Зависимость поведения от количества потоков

Количество параллельных потоков, создаваемых при выполнении приложения, в общем случае не является постоянной величиной. По умолчанию оно, как правило, равняется числу установленных на компьютере процессоров. Однако, число потоков может также задаваться программистом вручную (например, с помощью функции omp_set_num_threads, или выражения num_threads, которое имеет больший приоритет, чем эта функция). Помимо указанной функции и выражения существует еще и переменная среды OMP_NUM_THREADS, имеющая наименьший приоритет. Следовательно, число потоков является весьма ненадежным числом, к тому же значение этого числа по умолчанию может оказаться разным на разных компьютерах. Поведение вашего кода ни в коем случае не должно зависеть от количества выполняющих его потоков, если только вы не уверены до конца в том, что вам это действительно нужно.Рассмотрим пример некорректного кода, взятый из статьи [5]. Приведенная ниже программа должна по замыслу программиста вывести на экран все буквы английского алфавита.Некорректно:

```
omp_set_num_threads(4);
#pragma omp parallel private(i)
{
    int LettersPerThread = 26 / omp_get_num_threads();
    int ThisThreadNum = omp_get_thread_num();
    int StartLetter = 'a' + ThisThreadNum * LettersPerThread;
    int EndLetter = 'a' + ThisThreadNum * LettersPerThread + LettersPerThread;
    for (int i=StartLetter; i<EndLetter; i++)
          printf ("%c", i);
}
```

На практике, однако, будет выведено только 24 буквы из 26. Причина проблемы заключается в том, что 26 (число букв) делится на 4 (число потоков) с остатком. Следовательно, оставшиеся две буквы не будут выведены на экран. В качестве исправления можно либо отказаться от использования количества потоков в вычислениях и переписать код иначе, либо распределить вычисления на корректное число потоков (например, 2). Допустим, программист решил отказаться от использования количества потоков в своих вычислениях и возложить распределение работы между потоками на компилятор. В этом случае исправленная версия кода будет иметь следующий вид.Корректно:

```
omp_set_num_threads(4);
#pragma omp parallel for
for (int i = 'a'; i <= 'z'; i++)
{
     printf ("%c", i);
}
```

Все итерации будут гарантированно выполнены. Если необходимо задать способ распределения итераций между потоками, можно воспользоваться выражением schedule. Распределением работы между потоками теперь занимается компилятор, и он не забудет про две "лишние" итерации. К тому же, код оказался намного короче и понятнее.

12. Некорректное использование динамического создания потоков

В OpenMP слово dynamic встречается в двух контекстах: в выражении schedule(dynamic) и в переменной среды OMP_DYNAMIC, что вносит некоторую путаницу. Важно понимать разницу между этими двумя случаями и не считать, что выражением schedule(dynamic) можно пользоваться, только если переменная OMP_DYNAMIC имеет значение true. На самом деле эти случаи никак друг с другом не связаны.

Выражение schedule(dynamic) означает, что поделенные на куски (chunks) итерации цикла будут распределяться между потоками динамически - освободившись, поток будет брать следующую "порцию". В частности, в предыдущем примере это означало бы, что каждый из четырех потоков выведет по 6 букв, после чего тот поток, который освободится первым, выведет последние 2 буквы.

Переменная OMP_DYNAMIC задает возможность динамического определения числа потоков компилятором. Проблема заключается в том, что эта переменная имеет больший приоритет, чем даже выражение num_threads. Следовательно, если выполнение кода как-то зависит от количества выполняющих его потоков, поведение может стать некорректным, и это - еще один аргумент в пользу того, чтобы не писать зависящий от количества потоков код.Опыт показывает, что в Visual Studio 2008 переменная OMP_DYNAMIC по умолчанию имеет значение false. Однако, нет никаких гарантий, что это не изменится в будущем. В спецификации OpenMP [3] сказано, что значение этой переменной зависит от конкретной реализации. Следовательно, если в предыдущем примере программист решил пойти по легкому пути и все же использовать в своих вычислениях число потоков, вместо того, чтобы переписывать код, ему следует позаботиться о том, что это число потоков будет именно таким, как он указал. В противном случае на четырехпроцессорной машине код начнет работать некорректно.

Корректно:

```
if (omp_get_dynamic())
  omp_set_dynamic(0);
omp_set_num_threads(2);
#pragma omp parallel private(i)
{
    int LettersPerThread = 26 / omp_get_num_threads();
    int ThisThreadNum = omp_get_thread_num();
    int StartLetter = 'a' + ThisThreadNum * LettersPerThread;
    int EndLetter = 'a' + ThisThreadNum * LettersPerThread + LettersPerThread;
    for (i=StartLetter; i<EndLetter; i++)
          printf ("%c", i);
}
```

13. Одновременное использование общего ресурса

Если бы в предыдущем примере вывод производился не по одной букве в произвольном порядке, а хотя бы по две, проявилась бы еще одна проблема параллельного программирования - одновременное использование общего ресурса (которым в данном случае является консольное окно). Рассмотрим слегка измененный пример из статьи [6].

Некорректно:

```
#pragma omp parallel num_threads(2)
{
    printf("Hello World\n");
}
```

Вопреки ожиданиям программиста, на двухпроцессорной машине этот код, скорее всего, выведет нечто вроде следующих двух строчек:

```
HellHell oo WorWlodrl
d
```

Причина такого поведения заключается в том, что операция вывода строки на экран не является атомарной. Следовательно, два потока будут выводить свои символы одновременно. Та же самая ситуация возникнет при использовании стандартного потока вывода cout, а также любого другого объекта, доступного обоим потокам.

Если возникает необходимость выполнить какое-либо действие, изменяющее состояние такого объекта, из двух потоков, необходимо позаботиться о том, чтобы это действие выполнялось только одним из потоков в каждый момент времени. Для этого можно использовать директиву critical, или блокировку (о том, что именно предпочтительнее с точки зрения производительности, будет рассказано ниже).

Корректно:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    {
          printf("Hello World\n");
    }
}
```

14. Незащищенный доступ к общей памяти

Эта ошибка описана в статье [1]. По своей сути она аналогична предыдущей - если несколько потоков одновременно изменяют значение переменной, результат может быть непредсказуемым. Однако эта ошибка все же рассматривается отдельно от предыдущей, потому что в этом случае решение будет несколько иным. Поскольку операция над переменной может быть атомарной, в целях оптимизации быстродействия лучше использовать директиву atomic. Подробные рекомендации по поводу того, какой именно способ лучше использовать для защиты общей памяти, будут приведены ниже.

Некорректно:

```
int a = 0;
#pragma omp parallel
{
    a++;
}
```

Корректно:

```
int a = 0;
#pragma omp parallel
{
    #pragma omp atomic
    a++;
}
```

Еще одно возможное решение - использовать выражение reduction. В этом случае каждый поток получит собственную копию переменной a, выполнит с ней все необходимые действия, а затем произведет указанную операцию, чтобы объединить получившиеся значения с исходным.

Корректно:

```
int a = 0;
#pragma omp parallel reduction(+:a)
{
    a++;
}
printf("a=%d\n", a);
```

При таком изменении кода в случае двух потоков будет выведена строка "a=2".

15. Использование директивы flush с указателем

Директива flush служит для того, чтобы потоки обновили значения общих переменных. Например, если один поток установил значение доступной обоим потокам общей переменной a, равное 1, это не гарантирует, что другой поток, обратившись к той же переменной, получит значение 1. Еще раз подчеркнем, что данная директива обновляет именно значения переменных. Если в коде имеется доступный обоим потокам указатель на какой-либо объект, вызов директивы flush обновит только значение этого указателя, но не состояние объекта. Более того, в стандарте OpenMP [3] явно сказано, что переменная-аргумент директивы flush не должна быть указателем.

Некорректно:

```
MyClass* mc = new MyClass();
#pragma omp parallel sections
{
    #pragma omp section
    {
          #pragma omp flush(mc)
          mc->myFunc();
          #pragma omp flush(mc)
```

```
    }
    #pragma omp section
    {
          #pragma omp flush(mc)
          mc->myFunc();
          #pragma omp flush(mc)
    }
}
```

На самом деле этот код содержит две ошибки - уже упоминавшуюся ранее одновременную работу с общим объектом и использование flush с указателем. Следовательно, если метод myFunc изменяет состояние объекта, результат выполнения этого кода будет непредсказуемым. Для того, чтобы исправить эти ошибки, достаточно избавиться от одновременного использования общего объекта. Дело в том, что директива flush неявно выполняется при входе в критическую секцию и при выходе из нее (об этом будет подробнее рассказано позднее).

Корректно:

```
MyClass* mc = new MyClass();
#pragma omp parallel sections
{
    #pragma omp section
    {
          #pragma omp critical
          {
                mc->myFunc();
          }
    }
    #pragma omp section
    {
          #pragma omp critical
          {
                mc->myFunc();
          }
    }
}
```

16. Отсутствие директивы flush

Согласно спецификации OpenMP [3], эта директива неявно выполняется во многих случаях (полный список этих случаев будет приведен ниже, в описании одной из ошибок производительности). Понадеявшись на это, программист может не использовать эту директиву там, где она действительно необходима. Директива flush не выполняется в следующих случаях:

- При входе в параллельную секцию директивы for.
- При входе и при выходе из секции директивы master.
- При входе в параллельную секцию директивы sections.
- При входе в секцию директивы single.
- При выходе из секции директивы for, single или sections, если к директиве применено выражение nowait (вместе с неявным барьером эта директива убирает и неявную директиву flush).

Некорректно:

```
int a = 0;
#pragma omp parallel num_threads(2)
{
    a++;
    #pragma omp single
    {
          cout << a << endl;
    }
}
```

Корректно:

```
int a = 0;
#pragma omp parallel num_threads(2)
```

```
{
    a++;
    #pragma omp single
    {
          #pragma omp flush(a)
          cout << a << endl;
    }
}
```

Вторая версия кода учитывает необходимость применения директивы flush, однако, она тоже не идеальна - ей не хватает синхронизации.

17. Отсутствие синхронизации

Помимо необходимости применения директивы flush, программист должен помнить еще и о синхронизации потоков, не полагаясь при этом на OpenMP.

Исправленная версия кода из предыдущего примера отнюдь не гарантирует, что в консольное окно будет выведено число "2". Да, выполняющий секцию поток выведет значение переменной a, актуальное на момент вывода. Однако, нет никакой гарантии того, что потоки войдут в секцию single одновременно. В общем случае может быть взято как значение "1", так и "2". Такое поведение вызвано отсутствием синхронизации потоков. Директива single означает лишь то, что соответствующая секция должна быть выполнена одним потоком. А этим потоком с равной вероятностью может оказаться и тот, который завершился первым. Тогда на экран будет выведена цифра "1". Аналогичная ошибка описана в статье [7].

Неявная синхронизация потоков в виде директивы barrier выполняется только при выходе из секции директивы for, single или sections, если к директиве не применено выражение nowait, которое отменяет неявную синхронизацию. Во всех остальных случаях программист должен заботиться о синхронизации сам.

Корректно:

```
int a = 0;
#pragma omp parallel num_threads(2)
{
    #pragma omp atomic
    a++;
    #pragma omp barrier
    #pragma omp single
    {
          cout<<a<<endl;
    }
}
```

Эта версия кода является полностью корректной - на экран всегда будет выведена цифра "2". Отметим, что в этой версии кода директива flush отсутствует - она неявно включена в директиву barrier.

Теперь рассмотрим еще один весьма интересный пример отсутствия синхронизации, взятый из MSDN [8].

Некорректно:

```
struct MyType
{
    ~MyType();
};
MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {
      ...
    }
}
```

Проблема здесь заключается в том, что при завершении параллельной секции не делается синхронизация. В результате, на момент завершения процесса некоторые из потоков могут еще существовать, и они не получат извещения о том, кто процесс завершен. Реально деструктор для переменной

threaded_var будет вызван только в главном потоке. Поскольку эта переменная является параметром директивы threadprivate, ее копии, созданные в других потоках, не будут уничтожены, и возникнет утечка памяти. Чтобы этого избежать, необходимо реализовать синхронизацию вручную.

Корректно:

```
struct MyType
{
    ~MyType();
};
MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {
          ...
          #pragma omp barrier
    }
}
```

18. Внешняя переменная задана как threadprivate не во всех модулях

Начиная с этой ошибки, мы переходим к самому неприятному виду ошибок - к ошибкам, связанным с моделью памяти OpenMP. В принципе, ошибки, связанные с одновременным доступом к общим переменным, тоже можно было бы отнести к этой категории, поскольку они касаются переменных, являющихся аргументами выражения shared (все глобальные переменные в OpenMP считаются shared по умолчанию).

Прежде чем перейти к рассмотрению конкретных ошибок отметим, что практически все они, так или иначе, связаны с переменными, являющимися параметрами выражения private (а также его вариаций - директивы threadprivate и выражений firstprivate и lastprivate). Большинство этих ошибок можно избежать, если отказаться от директивы threadprivate и выражения private, и вместо этого просто объявлять соответствующие переменные как локальные переменные в параллельных секциях.

Теперь, когда вы предупреждены, перейдем к рассмотрению конкретных ошибок. Начнем с директивы threadprivate. Эта директива применяется, как правило, к глобальным переменным, в том числе и к внешним переменным, объявленным в другом модуле. В этом случае директива должна быть применена к переменной во всех модулях, в которых она встречается. Это правило описано в уже упомянутой выше статье MSDN [8].

Частным случаем этого правила является другое правило, приведенное в той же статье: директиву threadprivate нельзя применять к переменным, объявленным в динамически подключаемой библиотеке, которая будет загружаться с помощью функции LoadLibrary или ключа линковщика /DELAYLOAD (в этом случае функция LoadLibrary используется неявно).

19. Неинициализированные локальные переменные

При входе в поток для переменных, являющиеся параметрами директивы threadprivate, а также выражений private и lastprivate, создаются локальные копии. Эти копии являются неинициализированными по умолчанию. Следовательно, любая попытка работы с ними без предварительной инициализации приведет к ошибке во время выполнения программы.

Некорректно:

```
int a = 0;
#pragma omp parallel private(a)
{
   a++;
}
```

Корректно:

```
int a = 0;
#pragma omp parallel private(a)
```

```
{
    a = 0;
    a++;
}
```

Синхронизация и директива flush в данном случае не нужны, так как каждый поток обладает собственной копией переменной.

20. Забытая директива threadprivate

Поскольку директива threadprivate применяется к переменной лишь единожды, и обычно она используется для глобальных переменных, объявленных в начале модуля, про нее легко забыть - например, когда возникает необходимость что-то изменить в большом модуле, созданном полгода назад. В результате глобальная переменная, вопреки ожиданиям программиста, может оказаться не общей, как должно быть по умолчанию, а локальной для потока. Согласно спецификации OpenMP [3], значения таких переменных после соответствующей параллельной секции являются непредсказуемыми.

Некорректно:

```
int a;
#pragma omp threadprivate(a)
int _tmain(int argc, _TCHAR* argv[])
{
    ...
    a = 0;
    #pragma omp parallel
    {
            #pragma omp sections
            {
                  #pragma omp section
                  {
                        a += 3;
                  }
                  #pragma omp section
                  {
                        a += 3;
                  }
            }
            #pragma omp barrier
    }
    cout << "a = " << a << endl;
}
```

Поведение приведенной выше программы будет полностью соответствовать документации - иногда в консольное окно будет выводиться число "6" - то самое значение, которое ожидает программист, а иногда - "0", что, в принципе, более логично, так как именно это значение было присвоено переменной до входа в параллельную секцию. Теоретически такое же поведение должно наблюдаться, если вместо директивы threadprivate режим доступа к переменной "a" будет определяться выражениями private или firstprivate. Но на практике с компилятором среды Visual Studio 2008 описанное поведение удалось воспроизвести только с директивой threadprivate, именно поэтому этот пример и приведен в данной статье. К тому же такой вариант наиболее вероятен. Тем не менее, это не означает, что в других реализациях поведение будет корректным, поэтому следует учитывать и другие два варианта.

К сожалению, посоветовать хорошее решение в данном случае трудно, так как отказ от директивы threadprivate изменит поведение кода в других участках программы, а передать переменную, указанную в этой директиве, в выражение shared нельзя по правилам синтаксиса OpenMP. Единственное, что можно посоветовать в этой ситуации - использовать другую переменную.

Корректно:

```
int a;
#pragma omp threadprivate(a)
int _tmain(int argc, _TCHAR* argv[])
{
    ...
    a = 0;
    int b = a;
    #pragma omp parallel
```

```
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                b += 3;
            }
            #pragma omp section
            {
                b += 3;
            }
        }
        #pragma omp barrier
    }
    a = b;
    cout << "a = " << a << endl;
}
```

В этой версии кода переменная "a" как бы становится общей на время выполнения определенной параллельной секции. Конечно, это решение не является лучшим, но за то оно гарантирует, что старый код не изменит своего поведения.

В общем случае начинающим программистам рекомендуется использовать выражение default(none), которое заставит явно указывать режим доступа ко всем используемым в параллельной секции переменным. Это, конечно, потребует написания большего количества кода, но в результате удастся избежать многих ошибок, к тому же код станет более понятным.

21. Забытое выражение private

Возьмем сценарий, аналогичный предыдущему случаю - программисту требуется изменить что-то в модуле, написанном некоторое время назад, и выражение, определяющее режим доступа к переменной, находится достаточно далеко от того фрагмента кода, который должен быть изменен.

Некорректно:

```
int a;
#pragma omp parallel private(a)
{
...
a = 0;
#pragma omp for
for (int i = 0; i < 10; i++)
{
    #pragma omp atomic
    a++;
}
#pragma omp critical
{
    cout << "a = " << a;
}
}
```

На первый взгляд может показаться, что эта ошибка эквивалентна предыдущей, но это не так. В предыдущем случае вывод производился после параллельной секции, а здесь значение переменной берется внутри параллельной секции. В результате, если значение переменной перед циклом равно нулю, в случае двухпроцессорной машины, вместо ожидаемого числа "10" в консольное окно будет выведено число "5", потому что работа будет разделена пополам между потоками. Следовательно, каждый поток увеличит свое локальное значение переменной a лишь пять раз вместо ожидаемых десяти раз. Более того, значение переменной, получается, будет зависеть от количества потоков, выполняющих код параллельной секции. Кстати, эта ошибка возникнет и в случае использования выражения firstprivate вместо private.

Возможные пути исправления ошибки аналогичны предыдущей ситуации - нужно либо существенно переделывать весь старый код, либо подгонять поведение нового кода под поведение старого. В данном случае второе решение получается более элегантным, чем в предыдущем.

Корректно:

```
int a;
#pragma omp parallel private(a)
{
...
a = 0;
#pragma omp parallel for
for (int i = 0; i < 10; i++)
{
    #pragma omp atomic
    a++;
}
#pragma omp critical
{
    cout << "a = " << a;
}
}
```

22. Некорректное распараллеливание работы с локальными переменными

Данная ошибка аналогична предыдущей и также является, по сути, противоположностью ошибки "Ненужное разделение на потоки". В данном случае, однако, к ошибке может привести несколько иной сценарий.

Некорректно:

```
int a;
#pragma omp parallel private(a)
{
    a = 0;
    #pragma omp barrier
    #pragma omp sections
    {
          #pragma omp section
          {
                #pragma omp atomic
                a+=100;
           }
           #pragma omp section
           {
                #pragma omp atomic
                a+=1;
           }
    }
    #pragma omp critical
{
    cout << "a = " << a << endl;
}
}
```

В данном случае программист хотел увеличить значение локальной копии переменной в каждом из потоков на 101 и для этого использовал директиву sections. Однако, поскольку в директиве отсутствовало слово parallel, дополнительного распараллеливания не произошло. Работа распределилась на те же потоки. В результате на двухпроцессорной машине один поток выведет "1", а другой выведет "100". При большем количестве потоков результаты будут еще более неожиданными для программиста. Отметим, что если бы переменная "a" не была локальной (private), такой код был бы вполне корректным. Однако в случае локальной переменной секции необходимо распараллелить дополнительно.

Корректно:

```
int a;
#pragma omp parallel private(a)
{
    a = 0;
    #pragma omp barrier
    #pragma omp parallel sections
    {
```

```
            #pragma omp section
            {
                  #pragma omp atomic
                  a+=100;
            }
            #pragma omp section
            {
                  #pragma omp atomic
                  a+=1;
            }
    }
    #pragma omp critical
{
    cout<<"a = "<<a<<endl;
}
}
```

23. Неосторожное применение lastprivate

Напомним, что эта директива после параллельной секции присваивает переменной значение из лексически последней секции, либо из последней итерации цикла. Согласно спецификации, если переменной не присваивается значение в упомянутом фрагменте кода, ее значение после окончания соответствующей параллельной секции является неопределенным. Рассмотрим пример, аналогичный предыдущему.

Некорректно:

```
int a = 1;
#pragma omp parallel
{
    #pragma omp sections lastprivate(a)
    {
          #pragma omp section
          {
                ...
                a = 10;
          }
          #pragma omp section
          {
                ...
          }
    }
#pragma omp barrier
}
```

Такой код теоретически может привести к ошибке. На практике это воспроизвести не удалось. Однако, это не значит, что ошибка не возникнет никогда.

Если уж программисту действительно необходимо использовать выражение lastprivate, он должен четко представлять себе, какое именно значение будет присвоено переменной после выполнения параллельной секции. Ведь в общем случае ошибка может заключаться и в том, что переменной после параллельной секции будет присваиваться не то значение, которое ожидает программист. Например, программист будет ожидать, что переменной присвоится значение из того потока, который выполнится последним, а не из того, который является последним лексически. Для исправления подобных ошибок достаточно просто поменять код секций местами.

Корректно:

```
int a = 1;
#pragma omp parallel
{
    #pragma omp sections lastprivate(a)
    {
          #pragma omp section
          {
                ...
          }
          #pragma omp section
```

```
            {
                  ...
                  a = 10;
            }
    }
#pragma omp barrier
}
```

24. Непредсказуемые значения threadprivate-переменных в начале параллельных секций

Эта ошибка описана в спецификации OpenMP [3]. Если изменить значение переменной, объявленной как threadprivate, начальное значение переменной в параллельной секции окажется непредсказуемым.

К сожалению, пример, приведенный в спецификации, не компилируется в Visual Studio, потому что компилятор этой среды не поддерживает динамическую инициализацию переменных, фигурирующих в директиве threadprivate. Поэтому приведем свой собственный, менее сложный, пример.

Некорректно:

```
int a = 5;
#pragma omp threadprivate(a)
int _tmain(int argc, _TCHAR* argv[])
{
...
a = 10;
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    {
          printf("\nThread #%d: a = %d", omp_get_thread_num(),a);
    }
}
getchar();
return 0;
}
```

В результате выполнения этого кода один из потоков выведет значение 5, а другой выведет 10. Если убрать инициализацию переменной a до директивы threadprivate, один из потоков начнет выводить 0, а второй - по-прежнему 10. Избавиться от непредсказуемого поведения удастся лишь убрав второе присваивание. В этом случае оба потока будут выводить значение 5 (если первое присваивание все же оставить). Конечно, такие исправления изменят поведение кода. Они описаны здесь лишь для того, чтобы более четко показать поведение OpenMP в данном случае. Вывод здесь может быть только один: никогда не полагайтесь на компилятор в вопросах инициализации локальных переменных. В случае private и lastprivate попытка использования неинициализированных переменных вызовет уже описанную ранее ошибку во время выполнения программы, которую, по крайней мере, можно относительно легко локализовать. Но директива threadprivate, как видите, может давать непредсказуемые результаты без всяких ошибок. Вообще от использования этой директивы лучше всего отказаться. В этом случае код станет намного более предсказуемым и понятным.

Корректно:

```
int a = 5;
int _tmain(int argc, _TCHAR* argv[])
{
...
a = 10;
#pragma omp parallel num_threads(2)
{
    int a = 10;
    #pragma omp barrier
    #pragma omp critical
    {
          printf("\nThread #%d: a = %d", omp_get_thread_num(),a);
    }
```

```
}
getchar();
return 0;
}
```

25. Некоторые ограничения локальных переменных

В спецификации OpenMP приведено множество ограничений, касающихся локальных переменных. Некоторые из этих ограничений проверяются компилятором автоматически, однако, есть и такие, которые не проверяются:

- Переменная в выражении private не должна иметь ссылочный тип.
- Если переменная в выражении lastprivate является экземпляром класса, в этом классе должен быть определен конструктор копирования.
- Переменная в выражении firstprivate не должна иметь ссылочный тип.
- Если переменная в выражении firstprivate является экземпляром класса, в этом классе должен быть определен конструктор копирования.
- Переменная в выражении threadprivate не должна иметь ссылочный тип.

Фактически, это два требования: а) переменная не должна быть указателем б) если переменная является экземпляром класса, для нее должен быть определен конструктор копирования. Причины этих ограничений вполне очевидны - если переменная будет указателем, каждый поток получит по локальной копии этого указателя, и в результате все потоки будут работать через него с общей памятью. Ограничение насчет конструктора копирования тоже весьма логично - если в классе имеется поле, хранящее ссылку, корректно скопировать такой объект почленно не получится и в результате все потоки будут, опять же, работать с той же самой памятью.

Пример в данном случае займет много места и вряд ли необходим. Достаточно лишь запомнить общее правило. Если требуется создать локальную копию объекта, массива или какой бы то ни было области памяти, адресуемой по ссылке, то соответствующая ссылка должна оставаться общей переменной. Делать ее локальной бессмысленно. Содержимое памяти нужно либо копировать явно, либо (в случае объектов) доверить это компилятору, который использует конструктор копирования.

26. Локальные переменные не помечены как таковые

Данная ошибка описана в статье [1]. Суть ее заключается в том, что переменная, которая по замыслу программиста должна была быть локальной, не была помечена как локальная и, следовательно, используется как общая, поскольку этот режим доступа применяется ко всем переменным по умолчанию.

Для диагностики этой ошибки рекомендуется уже упоминавшееся выше использование выражения default(none).

Как видно, данная ошибка весьма обобщенная, и привести конкретный пример достаточно трудно. Однако, в одной из статьей [9] описана ситуация, в которой эта ошибка проявляется вполне явно.

Некорректно:

```
int _tmain(int argc, _TCHAR* argv[])
{
 const size_t arraySize = 100000;
 struct T {
   int a;
   size_t b;
 };
 T array[arraySize];
 {
   size_t i;
   #pragma omp parallel sections num_threads(2)
   {
     #pragma omp section
     {
```

```
        for (i = 0; i != arraySize; ++i)
          array[i].a = 1;
      }
      #pragma omp section
      {
        for (i = 0; i != arraySize; ++i)
          array[i].b = 2;
      }
    }
  }
  size_t i;
  for (i = 0; i != arraySize; ++i)
  {
    if (array[i].a != 1 || array[i].b != 2)
    {
      _tprintf(_T("OpenMP Error!\n"));
      break;
    }
  }
  if (i == arraySize)
    _tprintf(_T("OK!\n"));
     getchar();
     return 0;
}
```

Смысл программы очень прост - массив записей с двумя полями заполняется из двух потоков. Один поток заполняет одно поле записей, другой - другое. Потом делается проверка - все ли записи заполнены правильно.

Ошибка здесь заключается в том, что в цикле для счетчика используется общая переменная i. В результате при выполнении программы в некоторых случаях будет выводиться сообщение "OpenMP Error!", в других - возникать ошибка access violation, и лишь изредка будет выводиться "OK!". Для того, чтобы исправить эту ошибку, достаточно объявить переменную-счетчик как локальную переменную.

Корректно:

```
...
    #pragma omp parallel sections num_threads(2)
    {
      #pragma omp section
      {
        for (size_t i = 0; i != arraySize; ++i)
          array[i].a = 1;
      }
      #pragma omp section
      {
        for (size_t i = 0; i != arraySize; ++i)
          array[i].b = 2;
      }
    }
  }
...
```

В статье [1] имеется аналогичный пример, касающийся именно циклов for (он выделен как отдельная ошибка). Там сказано, что переменная-счетчик для цикла for, к которому применяется директива OpenMP for, должна быть объявлена как локальная. С первого взгляда может показаться, что там рассматривается абсолютно такая же ситуация, однако, это не так.

Дело в том, что, согласно станадрту OpenMP, переменная, использующаяся в качестве счетчика в цикле, неявно преобразуется из общей в локальную, даже если она является параметром выражения shared. Компилятор, делая это преобразование, не выдает никаких предупреждений. Именно этот случай описан в статье [1] и в этой ситуации преобразование действительно делается. Однако, в нашем примере вместо директивы for используется директива sections, и такое преобразование не производится.

Вывод из этих двух примеров один: переменная, использующаяся как счетчик в цикле внутри параллельной секции никогда не должна быть общей. Даже если на цикл распространяется действие директивы for, полагаться на неявное преобразование все равно не стоит.

27. Параллельная работа с массивом без упорядочивания итераций

Во всех предыдущих примерах в циклах for, распределенных на несколько потоков, не использовалась директива ordered (за исключением примера, в котором рассматривался непосредственно синтаксис этой директивы). Причина этого заключается в том, что во всех этих примерах порядок выполнения итераций был для нас несущественен. Однако, существуют ситуации, в которых эта директива необходима. В частности, она необходима, если выполнение одной итерации как-либо зависит от результата выполнения предыдущих итераций (эта ситуация описана в статье [6]). Рассмотрим пример.

Некорректно:

```
int* arr = new int[10];
for(int i = 0; i < 10; i++)
    arr[i] = i;
#pragma omp parallel for
for (int i = 1; i < 10; i++)
    arr[i] = arr[i - 1];
for(int i = 0; i < 10; i++)
    printf("\narr[%d] = %d", i, arr[i]);
```

Теоретически эта программа должна вывести последовательность из нулей. Однако, на двухпроцессорной машине будет выведено некоторое количество нулей и некоторое количество пятерок (это связано с тем, что итерации как правило делятся между потоками пополам). Проблема легко решается с помощью директивы ordered.

Корректно:

```
int* arr = new int[10];
for(int i = 0; i < 10; i++)
    arr[i] = i;
#pragma omp parallel for ordered
for (int i = 1; i < 10; i++)
{
    #pragma omp ordered
    arr[i] = arr[i - 1];
}
for(int i = 0; i < 10; i++)
    printf("\narr[%d] = %d", i, arr[i]);
```

Ошибки производительности

1. Ненужная директива flush

Все рассмотренные ранее ошибки влияли на поведение программы и являлись в той или иной мере критическими. Теперь же рассмотрим ошибки, которые повлияют лишь на производительность программы, не затрагивая логики ее работы. Эти ошибки описаны в статье[1].

Как уже упоминалось ранее, директива flush во многих случаях выполняется неявно. Следовательно, явное использование flush в этих случаях будет лишним. Лишняя директива flush, особенно если соответствующие переменные не будут указаны в качестве параметров директивы (в этом случае делается синхронизация всей общей памяти), может существенно замедлить выполнение программы.

Приведем случаи, в которых эта директива присутствует неявно и использовать ее нет нужды:

- В директиве barrier
- При входе и при выходе из параллельной секции директивы critical
- При входе и при выходе из параллельной секции директивы ordered
- При входе и при выходе из параллельной секции директивы parallel
- При выходе из параллельной секции директивы for
- При выходе из параллельной секции директивы sections
- При выходе из параллельной секции директивы single
- При входе и при выходе из параллельной секции директивы parallel for

- При входе и при выходе из параллельной секции директивы parallel sections

2. Использование критических секций или блокировок вместо atomic

Директива atomic работает быстрее, чем критические секции, поскольку некоторые атомарные операции могут быть напрямую заменены командами процессора. Следовательно, эту директиву желательно применять везде, где требуется защита общей памяти при элементарных операциях. К таким операциям, согласно спецификации OpenMP, относятся операции следующего вида:

- x binop= expr
- x++
- ++x
- x--
- --x

Здесь x - скалярная переменная, expr - выражение со скалярными типами, в котором не присутствует переменная x, binop - не перегруженный оператор +, *, -, /, &, ^, |, <<, или >>. Во всех остальных случаях применять директиву atomic нельзя (это проверяется компилятором).

Вообще, с точки зрения убывания быстродействия, средства защиты общих данных от одновременной записи располагаются так: atomic, critical, omp_set_lock.

3. Ненужная защита памяти от одновременной записи

Любая защита памяти от одновременной записи замедляет выполнение программы, будь то атомарная операция, критическая секция или блокировка. Следовательно, в тех случаях, когда она не нужна, эту защиту лучше не использовать.

Переменную не нужно защищать от одновременной записи в следующих случаях:

- Если переменная является локальной для потока (а также, если она участвует в выражении threadprivate, firstprivate, private или lastprivate)
- Если обращение к переменной производится в коде, который гарантированно выполняется только одним потоком (в параллельной секции директивы master или директивы single).

4. Неоправданно большое количество кода в критической секции

Критические секции замедляют выполнение программы. Во-первых, из-за критических секций потокам приходится ждать друг друга, а это уменьшает приращение производительности, достигнутое благодаря распараллеливанию кода. Во-вторых, на вход в критические секции и на выход из них также затрачивается некоторое время.

Следовательно, применение критических секций там, где они не нужны, нежелательно. В критические секции не рекомендуется помещать вызовы сложных функций, а также код, не работающий с общими переменными, объектами или ресурсами. Дать конкретные рекомендации тут достаточно трудно - в каждом случае программист должен сам определить, какой код вносить в критическую секцию, а какой - не вносить.

5. Слишком частое применение критических секций

Как уже упоминалось в предыдущем пункте, вход в критическую секцию и выход из нее требуют определенного времени. Следовательно, частые входы в критическую секцию могут существенно за-

медлить программу. Чтобы избежать этого, рекомендуется насколько возможно снижать число входов в критические секции. Рассмотрим несколько измененный пример из статьи [1].

Некорректно:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i )
{
    #pragma omp critical
    {
          if (arr[i] > max) max = arr[i];
    }
}
```

Очевидно, что если вынести условие из критической секции, вход в нее будет производиться далеко не во всех итерациях цикла.

Корректно:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i )
{
    #pragma omp flush(max)
    if (arr[i] > max)
    {
           #pragma omp critical
           {
               if (arr[i] > max) max = arr[i];
           }
    }
}
```

Такое простое исправление может существенно увеличить производительность вашего кода, и им не стоит пренебрегать.

Заключение

На момент написания эта статья является наиболее полным списком возможных ошибок при программировании с использованием OpenMP, собранных из различных источников и из личного опыта авторов. Еще раз напомним, что все эти ошибки не диагностируются стандартными компиляторами. Подводя итоги, приведем краткое описание всех ошибок с соответствующими выводами, которые должен сделать из них программист.

Ошибка	Вывод
1. Отсутствие /openmp	При создании проекта нужно сразу же включить соответствующую опцию.
2. Отсутствие parallel	Необходимо тщательно следить за синтаксисом используемых директив.
3. Отсутствие omp	Необходимо тщательно следить за синтаксисом используемых директив.
4. Отсутствие for	Необходимо тщательно следить за синтаксисом используемых директив.
5. Ненужное распараллеливание	Необходимо тщательно следить за синтаксисом используемых директив и четко представлять себе их назначение.
6. Неправильное применение ordered	Необходимо тщательно следить за синтаксисом используемых директив.
7. Переопределение количества потоков внутри параллельной секции	Количество потоков нельзя изменять внутри параллельной секции.
8. Попытка использовать блокировку без инициализации переменной	Переменная, использующаяся для блокировки, должна быть обязательно инициализирована функцией omp_init_lock.
9. Попытка снять блокировку не	Каждый из потоков, использующих блокировку, должен содержать

из того потока, который ее установил	вызов как блокирующей (omp_set_lock, omp_test_lock), так и разблокирующей (omp_unset_lock) функции.
10. Попытка использования блокировки как барьера	Каждый из потоков, использующих блокировку, должен содержать вызов как блокирующей (omp_set_lock, omp_test_lock), так и разблокирующей (omp_unset_lock) функции.
11. Зависимость поведения от количества потоков	Поведение вашего кода не должно зависеть от числа исполняющих его потоков.
12. Некорректное использование динамического создания потоков	Если вам все же необходим код, зависящий от числа потоков, вы должны убедиться в том, что он будет выполняться именно нужным числом потоков (для этого следует явно отключать динамическое создание потоков). Вообще использовать динамическое создание потоков не рекомендуется.
13. Одновременное использование общего ресурса	Одновременный доступ к общему ресурсу должен быть защищен критической секцией или блокировкой.
14. Незащищенный доступ к общей памяти	Одновременный доступ к общей памяти должен быть защищен как атомарная операция (наиболее предпочтительно), критической секцией, или блокировкой.
15. Использование директивы flush с указателем	Применять директиву flush к указателю бессмысленно - при этом обновится значение указателя, но не памяти, на которую он ссылается.
16. Отсутствие директивы flush	Отсутствие директивы flush может привести к чтению или записи некорректных данных.
17. Отсутствие синхронизации	Отсутствие синхронизации также может привести к чтению или записи некорректных данных.
18. Внешняя переменная задана как threadprivate не во всех модулях	Если переменная, фигурирующая в директиве threadprivate, является внешней, она должна быть объявлена как threadprivate во всех модулях, в которых она встречается. Вообще от использования директивы threadprivate и выражений private, firstprivate, lastprivate лучше отказаться. Вместо этого рекомендуется объявлять локальные переменные в коде параллельной секции, а соответствующие начальные и конечные присваивания (если они необходимы) производить с общей переменной.
19. Неинициализированные локальные переменные	По умолчанию локальные переменные, фигурирующие в выражениях private и lastprivate являются неинициализированными. Брать их значения без предварительной инициализации нельзя. Вообще от использования директивы threadprivate и выражений private, firstprivate, lastprivate лучше отказаться. Вместо этого рекомендуется объявлять локальные переменные в коде параллельной секции, а соответствующие начальные и конечные присваивания (если они необходимы) производить с общей переменной.
20. Забытая директива threadprivate	Забытая директива threadprivate может повлиять на поведение всего модуля. Вообще от использования директивы threadprivate и выражений private, firstprivate, lastprivate лучше отказаться. Вместо этого рекомендуется объявлять локальные переменные в коде параллельной секции, а соответствующие начальные и конечные присваивания (если они необходимы) производить с общей переменной.
21. Забытое выражение private	Необходимо четко контролировать режим доступа к переменным. Новичкам рекомендуется использовать выражение default(none), чтобы режим доступа всегда приходилось задавать явно. Вообще от использования директивы threadprivate и выражений private, firstprivate, lastprivate лучше отказаться. Вместо этого рекомендуется объявлять локальные переменные в коде параллельной секции, а

	соответствующие начальные и конечные присваивания (если они необходимы) производить с общей переменной.
22. Некорректное распараллеливание работы с локальными переменными	Если распараллелить выполнение кода, работающего с локальными переменными, на те же потоки, в которых они созданы, разные потоки получат разные значения переменных.
23. Неосторожное применение lastprivate	Нужно четко представлять себе, какое именно значение будет в итоге записано в переменную после выполнения параллельной секции, если переменная объявлена как lastprivate. Вообще от использования директивы threadprivate и выражений private, firstprivate, lastprivate лучше отказаться. Вместо этого рекомендуется объявлять локальные переменные в коде параллельной секции, а соответствующие начальные и конечные присваивания (если они необходимы) производить с общей переменной.
24. Непредсказуемые значения threadprivate-переменных в начале параллельных секций	Значение переменной, объявленной как threadprivate, является непредсказуемым в начале параллельной секции, особенно если переменной до этого присваивалось какое-либо значение. Вообще от использования директивы threadprivate и выражений private, firstprivate, lastprivate лучше отказаться. Вместо этого рекомендуется объявлять локальные переменные в коде параллельной секции, а соответствующие начальные и конечные присваивания (если они необходимы) производить с общей переменной.
25. Некоторые ограничения локальных переменных	Локальные переменные не должны быть ссылками (при этом через них будет производиться одновременный доступ к одной и той же общей памяти) и экземплярами классов, не имеющих конструктора копирования (если в классе содержатся ссылки, объект может быть скопирован некорректно).
26. Локальные переменные не помечены как таковые	Необходимо четко контролировать режим доступа к переменным. Новичкам рекомендуется использовать выражение default(none), чтобы режим доступа всегда приходилось задавать явно. В частности, переменная, использующаяся как счетчик в цикле, всегда должна быть локальной.
27. Параллельная работа с массивом без упорядочивания итераций	Если очередная итерация цикла при работе с массивом зависит от результата предыдущей, необходимо упорядочивать выполнение цикла с помощью директивы ordered.
1. Ненужная директива flush	Директиву flush нет смысла применять там, где она и без того включена по умолчанию.
2. Использование критических секций или блокировок вместо atomic	Для защиты элементарных операций лучше использовать директиву atomic. Использование критических секции и блокировок менее предпочтительно, поскольку оно замедляет работу программы.
3. Ненужная защита памяти от одновременной записи	Операцию работы с памятью нет смысла защищать, если речь идет о локальной переменной, либо если код гарантированно будет выполнять только один поток.
4. Неоправданно большое количество кода в критической секции	В критические секции стоит включать минимум кода. Не стоит включать в критические секции код, который не работает с общей памятью, и вызовы сложных функций.
5. Слишком частое применение критических секций	Количество входов в критические секции и выходов из них лучше всего сократить. Например, можно выносить условия из критических секций.

Таблица 1 - Краткий список основных ошибок.

Вообще все ошибки можно разделить на три основные категории:

- Незнание синтаксиса OpenMP.
- Непонимание принципов работы OpenMP.
- Некорректная работа с памятью (незащищенный доступ к общей памяти, отсутствие синхронизации, неправильный режим доступа к переменным, и т. п.).

Приведенный в этой статье список ошибок, конечно, не является полным. Существует множество других ошибок, не затронутых здесь. Возможно, более полные списки будут приведены в новых статьях по этой теме.

Как бы то ни было, все эти ошибки в большинстве случаев можно легко диагностировать автоматически, средствами статического анализатора. На данный момент диагностику некоторых (лишь очень немногих) из них выполняет Intel Thread Checker. Некоторые ошибки диагностируются компиляторами, отличными от компилятора Visual Studio. Однако специализированного инструмента пока не существует. В частности, Intel Thread Checker обнаруживает одновременный доступ к общим переменным, некорректное использование директивы ordered и отсутствие ключевого слова for в директиве #pragma omp parallel for [1].

Также полезной для разработчиков могла бы оказаться программа, визуально отображающая распараллеливание кода и режимы доступа к переменным в соответствующих параллельных секциях. Такой программы пока что также не существует.

В данный момент авторами начата работа над статическим анализатором кода (рабочее название VivaMP), который будет диагностировать все перечисленные выше ошибки и, возможно, некоторые другие. Этот анализатор сможет существенно упростить поиск ошибок при разработке параллельных приложений (напомним, что почти все эти ошибки воспроизводятся нестабильно) и их устранение. Дополнительную информацию вы можете получить на странице, посвященной проекту VivaMP (http://www.viva64.com/vivamp.php).