Бурова Ирина Герасимовна,
Демьянович Юрий Казимирович

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ УРОК 6. ОБ ИНТЕРФЕЙСЕ OPEN MP

Предыдущие статьи (см. [1—5]) были посвящены теории и практике распараллеливания: мы ознакомились с некоторыми проблемами распараллеливания, с теоретическими подходами к этой проблеме, с некоторыми коммуникационными средами и со способами распараллеливания последовательных программ с использованием интерфейса MPI. В данной статье рассмотрим распараллеливание с помощью интерфейса Open MP; версии этого интерфейса часто применяются на вычислительных системах с общей памятью и, в частности, на многоядерных компьютерах.

1. ВВЕДЕНИЕ В ТЕХНОЛОГИЮ OPEN MP

Стандарт Open MP разработан для языков Fortran и C, причем конструкции Open MP в различных языках мало отличаются.

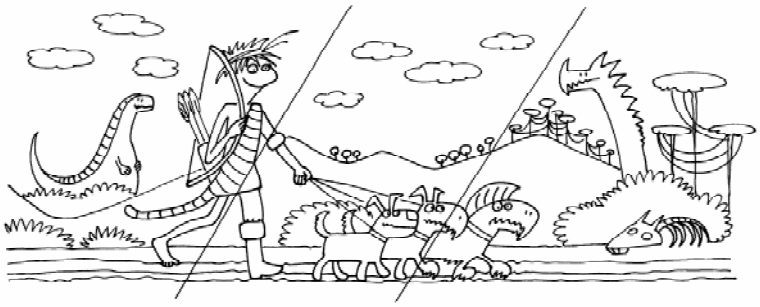
В технологии Open MP за основу берется последовательная программа. Распараллеливание программы состоит в том, что весь текст разбивается на последовательные и параллельные области. Сначала порождается *основная нить*, которая начинает выполнение программы, а также исполняет все последовательные секции программы.

При входе в параллельную область основная нить порождает *дополнительные нити*, которые нумеруются последовательными натуральными числами; основная нить получает номер 0. Все нити используют одну и ту же программу, причем они выполняют лишь ту ее часть, которая соответствует номеру нити.

В параллельной области все переменные делятся на два класса: *общие* (`shared`) и *локальные* (`private`). Общие переменные существуют в одном экземпляре и доступны всем нитям. Объявление локальной переменной приводит к порождению многих экземпляров этой переменной (по числу нитей) под одним именем – по одному экземпляру для каждой нити. Каждая нить может изменять значение своего экземпляра локальной переменной; это не влияет на значения экземпляров этой переменной в других нитях.

2. ДИРЕКТИВЫ OPEN MP

Для директив Open MP используется структура, предназначенная для коммента-



При входе в параллельную область основная нить порождает дополнительные нити...

риев, но она дополняется определенной комбинацией символов; например, на Фортране для этого можно использовать комбинации **!$OMP**, **C$OMP** или ***$OMP** (аналогичные комбинации имеются на языке Си).

Итак, описание параллельной области имеет вид

```
!$OMP    PARALLEL
     <параллельная область программы>
!$OMP    END PARALLEL
```

При этом все нити должны достигнуть директиву **!$OMP END PARALLEL**, и лишь после этого происходит слияние всех нитей в основную нить, которая и продолжает процесс.

Возможно создание параллельной области при выполнении некоторого условия:

```
!$OMP PARALLEL IF (<условие>);
```

при этом, если **<условие>** не выполнено, то эта директива пропускается. Для определения номеров нитей и их общего количества служат директивы

```
!$OMP   MP_GET_THREAD_NUM,
!$OMP   OMP_GET_NUM_THREADS.
```

Например, возможно использовать схему (см. листинг 1), где часть программы, находящаяся между **THEN** и **ELSE**, выполняется нитью с номером 3, а все остальные нити выполняют программу между **ELSE** и **ENDIF**.

3. РАСПАРАЛЛЕЛИВАНИЕ ЦИКЛОВ

Если в параллельной секции встретился оператор цикла без дополнительных указаний, то каждая нить выполнит все итерации данного цикла. Если требуется распределение итераций между нитями, то пишут

```
!$OMP     DO [опция]
     <do-цикл>
[!$OMP   END DO];
```

Здесь опцией (командой) служит **SCHEDULE** со следующими параметрами.

1. Параметр **STATIC [,m]** дает блочно-циклическое распределение итераций: первый блок из **m** итераций выполняет первая нить, второй блок из **m** итераций – вторая и т. д., а затем распределение начинается снова с первой нити.

2. Параметр **DYNAMIC [,m]** приводит к динамическому распределению итераций с фиксированным размером блока: сначала все нити получают порции по **m** итераций, а затем каждая нить, заканчивающая свою работу, получает порцию, содержащую **m** итераций.

3. Параметр **GUIDED [,m]** дает динамическое распределение итераций между блоками уменьшающегося размера: сначала блоки берутся максимальных размеров (определяемых реализацией Open MP), а затем эти размеры уменьшаются до тех пор, пока не достигнут значения **m**.

4. Наконец, использование параметра **RUNTIME** позволяет распределять итерации цикла в процессе работы программы, в зависимости от значения переменной **OMP_SCHEDULE**, задаваемой пользователем.

Например, использование строки **!$OMP DO SCHEDULE (DYNAMIC,10)** приводит к динамическому распределению итераций блоками по 10 итераций.

Циклы можно распараллеливать лишь в случае, если работа любой нити не зависит от работы остальных нитей и если нет побочных выходов из цикла.

Фрагмент программы

```
!$OMP DO SCHEDULE (STATIC,2)
       DO i=1,n
        DO i=1,m
        A(i,j)=(B(i,j-1)+b(i-1,j))/2.0
        END DO
       END DO
!$OMP END DO
```

Листинг 1

```
!$OMP    IF(OMP_GET_THREAD_NUM ().EQ.3) THEN
 <индивидуальный фрагмент программы для нити с номером 3>
!$OMP    ELSE
 <фрагмент программы для всех остальных нитей>
!$OMP     ENDIF,
```

приводит к блочно-циклическому распределению итераций (по две итерации в блоке) лишь во внешнем цикле; внутренний цикл будет выполняться полностью каждой нитью.

4. НЕЗАВИСИМЫЕ ФРАГМЕНТЫ

Можно использовать *параллелизм независимых фрагментов*:

```
!$OMP   SECTIONS
!$OMP   SECTION
         <фрагмент 1>
!$OMP   SECTION
         <фрагмент 2>
!$OMP   SECTION
         <фрагмент 3>
!$OMP   END SECTIONS;
```

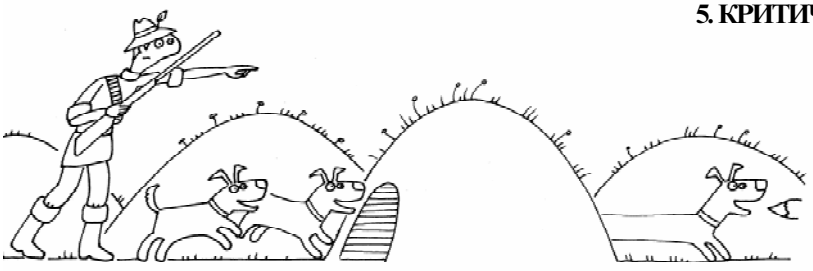
при этом фрагменты 1, 2, 3 разрешается выполнять параллельно.

Для выполнения фрагмента программы лишь одной нитью следует его поставить между директивами **!$OMP SINGLE** и **!$OMP END SINGLE**.

Рассмотрим следующую программу (см. листинг 2).

Каждая нить здесь выполняет фрагмент программы, печатая приветствие и номер нити; кроме этого, основная нить печатает общее число порожденных нитей.

Здесь переменные **NTHREADS** и **TID** объявлены локальными; однако первая (**NTHREADS**) могла бы быть объявлена общей, ибо она используется лишь в одной нити, при этом захватываемая память сократилась бы (исчезли бы ненужные копии).



При этом все нити должны достигнуть директиву ...,
и лишь после этого происходит слияние всех нитей в основную нить,
которая и продолжает процесс...

5. КРИТИЧЕСКИЕ СЕКЦИИ

Критическая секция в Open MP имеет вид как в листинге 3.

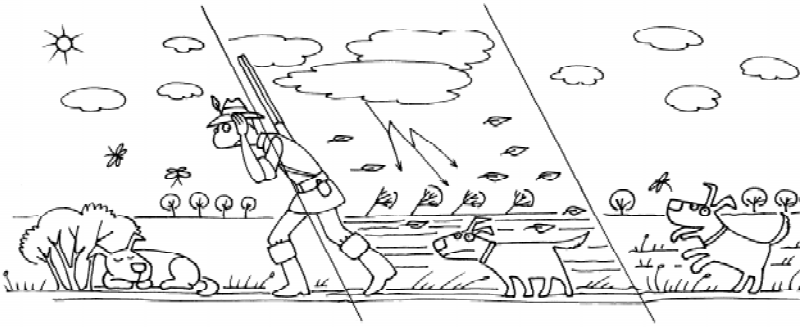
При этом все нити, выполнившие директиву **CRITICAL**, останавливаются и ждут, пока проходящая ее нить не завершит работу; после этого

Листинг 2

```
          PROGRAMM HELLO
          INTEGER NTHREADS, TID
C  Порождение нитей с локальными переменными
!$OMP  PARALLEL PRIVATE (NTHREADS,TID)
C  Получение и распечатка своего номера
          TID=OMP_GET_THREAD_NUM()
          PRINT*, "Hello World from thread=", TID
C  Участок кода для нити-мастера
          IF (TID.EQ 0) THEN
          NTHREADS= OMP_GET_NUM_THREADS()
          PRINT*, "NUMBER of threads=",NTHREADS
C  Завершение параллельной секции
!$OMP  END PARALLEL
          END
```

Листинг 3

```
!$OMP CRITICAL [(<имя критической секции >)]
                  <фрагмент программы >
!$OMP END CRITICAL [(<имя критической секции >)]
```



...все нити, выполнившие директиву CRITICAL, останавливаются и ждут, пока проходящая ее нить не завершит работу; после этого, в критическую секцию входит следующая нить...

в критическую секцию входит следующая нить.

Заметим, что критические секции следует использовать редко, ибо их применение приводит к последовательному (а не к параллельному) выполнению соответствующих участков программы.

Обратимся теперь к распараллеливанию в Vusual Studio на языке Си с помощью технологии OpenMP. В этом нам поможет методический материал [6].

6. УСЛОВНАЯ КОМПИЛЯЦИЯ НА ЯЗЫКЕ СИ

Для проверки того, что компилятор поддерживает какую-либо версию OpenMP, достаточно написать директивы условной компиляции **#ifdef**.

```
#include "stdafx.h"
#include <stdio.h>
int _tmain()
{
#ifdef _OPENMP
printf("OpenMP is supported!\n");
#endif
return 0;
}
```

7. ЗАМЕР ВРЕМЕНИ. РАБОТА С СИСТЕМНЫМИ ТАЙМЕРАМИ НА ЯЗЫКЕ СИ

В OpenMP предусмотрены функции для работы с системным таймером. Функция **omp_get_wtime()** возвращает в вызвавшем потоке астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы выбранного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменён за время существования процесса. Таймеры разных потоков могут быть не синхронизированы и выдавать различные значения.

Функция **omp_get_wtick()** возвращает в вызвавшем потоке разрешение таймера в секундах. Это время можно рассматривать как меру точности таймера.

Рассмотрим применение функций **omp_get_wtime()** и **omp_get_wtick()** для работы с таймерами в OpenMP. В приведенном ниже фрагменте производится замер начального времени, затем сразу происходит замер конечного времени. Разность этих значений даёт замер времени выполняемого фрагмента программы. Кроме того, измеряется точность системного таймера (см. листинг 4).

Операция создания потоков и операция синхронизации по трудоемкости эквивалентны выполнению примерно 1000 операций,

Функция ... возвращает в вызвавшем потоке астрономическое время в секундах..., прошедшее с некоторого момента в прошлом...

Листинг 4

```
#include "stdafx.h"
#include <stdio.h>
#include <omp.h>
int _tmain(int argc, char *argv[])
{
    double s_t, e_t, tick;
    s_t =omp_get_wtime();
    e_t = omp_get_wtime();
    tick =omp_get_wtick();
    printf("Замер времени в секундах %lf\n", e_t-s_t);
    printf("Точность таймера %lf\n", tick);
     return 0;
}
```

поэтому для эффективного распараллеливания вычислений требуется, чтобы трудоемкость соответствующего фрагмента программы была значительно больше 2000 операций.

Можно осуществить организацию **NUM_THREADS** потоков. При наличии достаточного количества свободных процессоров все потоки могут выполнять свою работу параллельно.

Не следует думать, что при организации N параллельных потоков программа станет работать в N раз быстрее. Это противоречит закону Амдала.

8. ЗАКОН АМДАЛА И ЕГО СЛЕДСТВИЯ

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f, где $0 \le f \le 1$ (при этом доля понимается не по статическому числу строк кода, а по числу операций в процессе выполнения).

Крайние случаи в значениях f соответствуют полностью параллельным ($f = 0$) и полностью последовательным ($f = 1$) программам.

Так вот, для того чтобы оценить, какое ускорение S может быть получено на компьютере из p процессоров при данном значении f, можно воспользоваться законом Амдала:

$$S \le \frac{1}{f + (1 - f)/p}.$$

9. ЗАПУСК ПОСЛЕДОВАТЕЛЬНОЙ ПРОГРАММЫ

Упражнение 1. Напишите последовательную программу, которая печатает слова «hello world».

```
void _tmain()
{
  int ID = 0;
 printf(" Hello(%d) ", ID);
 printf(" world(%d) \n", ID);
}
```

Начнем последовательное изучение основ техники распараллеливания вычислений со следующего замечания. Большинство конструкций в OpenMP – это директивы компилятору

```
#pragma omp construct[clause
                      [clause]…]
```

10. ЗАПУСК ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ

Для того чтобы фрагмент выполнялся в параллельном режиме, используем директиву

```
#pragma omp parallel.
```

Если нужно, чтобы вычисления проводились на четырех процессорах, применяйте директиву

```
#pragma omp parallel num_threads(4).
```

В начале программы подключаем **#include <omp.h>**.

Упражнение 2. Напишем многопотоковую программу, которая печатает слова «hello world».

```
#include "stdafx.h"
#include <omp.h>
#include <stdio.h>
   int _tmain()
 {
  #pragma omp parallel
  {
    int nid = 0;
   printf(" hello(%d) ", nid);
   printf(" world(%d) \n", nid);
  }
}}
```

Не забудем включить поддержку OpenMP в диалоговом окне свойств проекта\break («Configuration Properties | C/C++ | Language», более подробная информация находится в приложении).

Напомним, что распараллеливать цикл не всегда возможно. Например, если для выполнения итерации i нужно знать результат итерации $i-1$, то есть итерация i зависит от итерации $i-1$, то распараллелить невозможно. Например, при попытке распараллеливания цикла возникнет сообщение об ошибке

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

Прежде чем запустить на выполнене следующий фрагмент (см. листин 5), подумайте, какой может быть результат.

Ответ. В начале печатается «Последовательная область 1», затем по директиве parallel порождаются новые потоки, каждый из которых напечатает текст «Параллельная область», затем порождённые потоки объединяются и оставшийся поток-мастер напечатает текст «Последовательная область 2».

Теперь запустите на выполнение следующую программу (см. листинг 6) и объясните результат.

Листинг 5

```
#include <omp.h>
#include <stdio.h>
int _tmain(int argc, char *argv[])
{
   printf("Последовательная область 1\n");
  #pragma omp parallel
  {
      printf("Параллельная область\n");
  }
  printf("Последовательная область 2\n");
}
```

Листинг 6

```
#include "stdafx.h"
#include <omp.h>  // подключение OpenMP
#include <stdio.h>

 void _tmain()
  {
  #pragma omp parallel // начало параллельной области
    {
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
     } // конец параллельной области
  }
```

Литература

1. *Демьянович Ю.К.* Параллельные вычисления. Урок 1. Параллельная форма // Компьютерные инструменты в школе, 2011. № 1. С. 36–39.

2. *Демьянович Ю.К.* Параллельные вычисления. Урок 2. О средствах распараллеливания... Элементарная параллельная программа // Компьютерные инструменты в школе, 2011. № 2. С. 17–21.

3. *Демьянович Ю.К.* Параллельные вычисления. Урок 3. О проблемах распараллеливания // Компьютерные инструменты в школе, 2011. № 3. С. 35–41.

4. *Демьянович Ю.К.* Параллельные вычисления. Урок 4. Программируем на MPI // Компьютерные инструменты в школе, 2011. № 4. С. 30–39.

5. *Демьянович Ю.К.* Параллельные вычисления. Урок 5. Об архитектуре параллельных систем. Коммуникационные среды // Компьютерные инструменты в школе, 2011. № 5. С. 24–37.

6. *Антонов А.С.* Параллельное программирование с использованием технологии OpenMP. М.: Изд-во МГУ, 2009.

Демьянович Юрий Казимирович,
доктор физико-математических наук, профессор, заведующий кафедрой параллельных алгоритмов математико-механического факультета СПбГУ,

Бурова Ирина Герасимовна,
доктор физико-математических наук, профессор кафедры вычислительной математики СПбГУ.