Реализация параллельных шаблонов: OpenMP

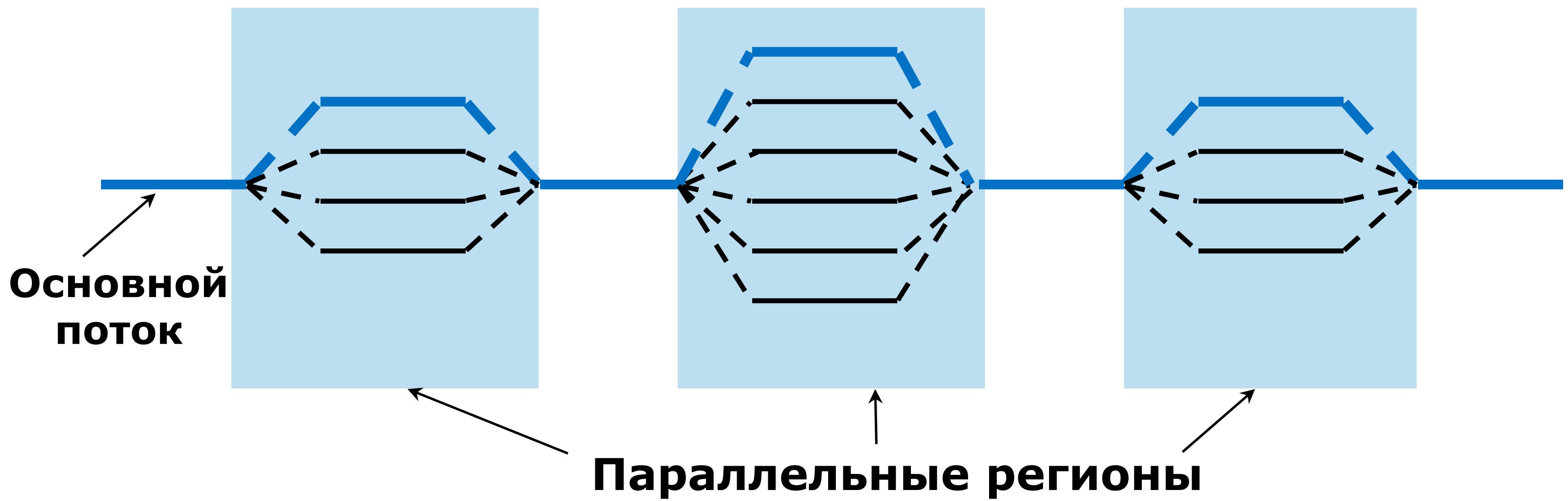
Алексей Федотов

SSG/DPD/TCAR/Threading Runtimes

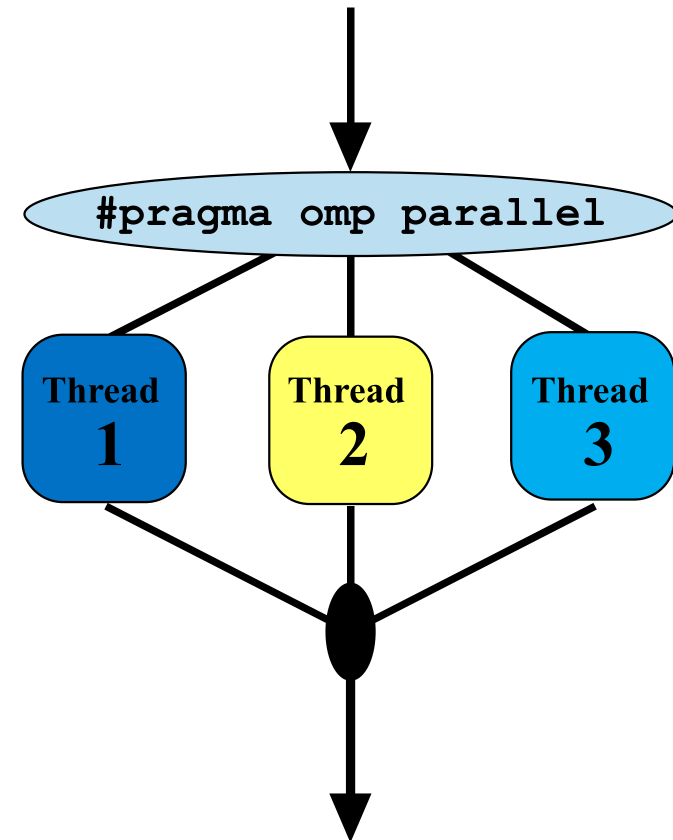
19.11.2015

Модель

- **Основной поток создаёт потоки по мере необходимости**
- **Параллелизм появляется в процессе исполнения: последовательная программа эволюционирует в параллельную**



Параллельные регионы



Определяет параллельный регион над структурированным участком кода

Потоки создаются в тот месте кода, где стоит указание «parallel»

Потоки блокируются в конце региона

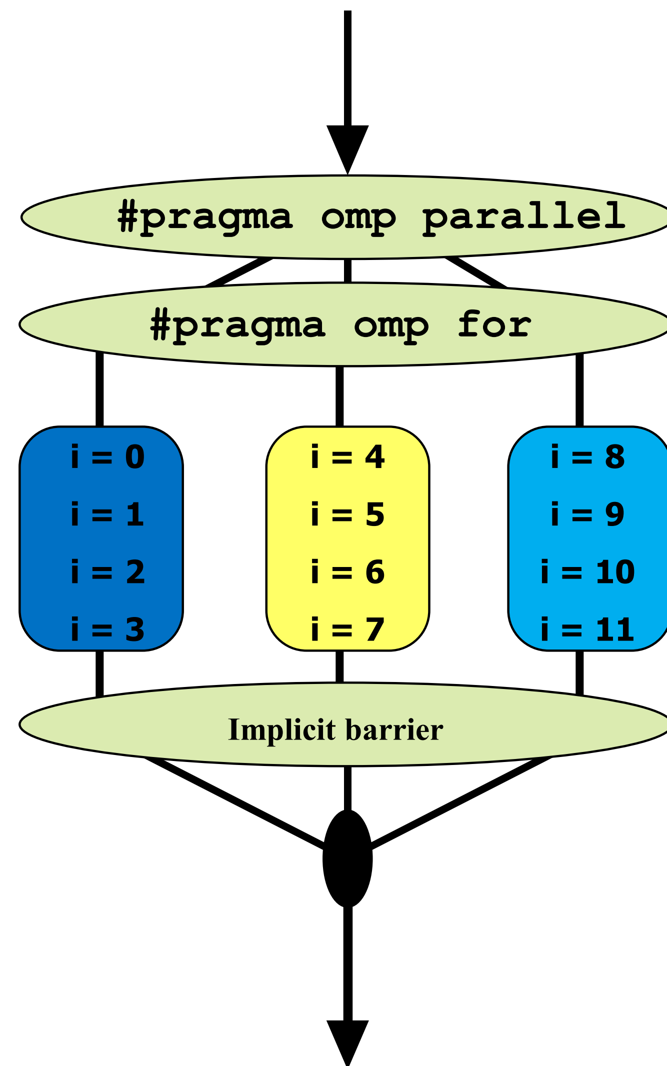
Данные становятся общими, если не специфицировано иное

C/C++ :

```
#pragma omp parallel
    {
        block
    }
```

Конструкция «Распределение работы»

```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; i++)
      c[i] = a[i] + b[i]
```



Потокам назначаются независимые итерации

Потоки ждут в конце

Комбинирование директив

Следующие сегменты кода эквивалентны

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) {
       res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0; i< MAX; i++) {
         res[i] = huge();
    }
```

Директива Private

Порождает <u>отдельную</u> переменную для каждого потока

- Переменные не инициализируются. C++ объекты конструируются по умолчанию
- Значение переменных после параллельного региона неопределено

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
        c[i] = x + y;
     }
}
```

Пример: скалярное произведение

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Всё верно?

Как делать корректно?

Ответственность за предоставление эксклюзивного доступа к разделяемой памяти лежит на пользователе:

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

Пример: редукция

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
```

Локальная копия переменной *sum* у каждого потока

Локальные копии складываются вместе, и сохраняются в глобальной переменной

intel

Поддеживаемые операции редукции (C/C++)

Целый набор ассоциативных и коммутативных операций может быть использован с директивой редукции

Начальные значения те, которые имею смысл.

Оператор	Начальное значение
+	0
*	1
-	0
^	0

<table>
<tr><th>Оператор</th><th>Начальное значение</th></tr>
<tr><td>&</td><td>~0</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
</table>

Распределение

Директива schedule	Когда следует использовать
static	Предсказуемое и схожее количество работы на каждой итерации
dynamic	Непредсказуемое, высоко изменчивое количество работы на каждой итерации
guided	Специальный случай динамического распределения. Призван сократить накладные расходы, связанные с самим распределением
auto	Решение о распределении принимается компилятором
runtime	Распределение задаётся переменной окружения OMP_SCHEDULE.

Пример использования директивы schedule

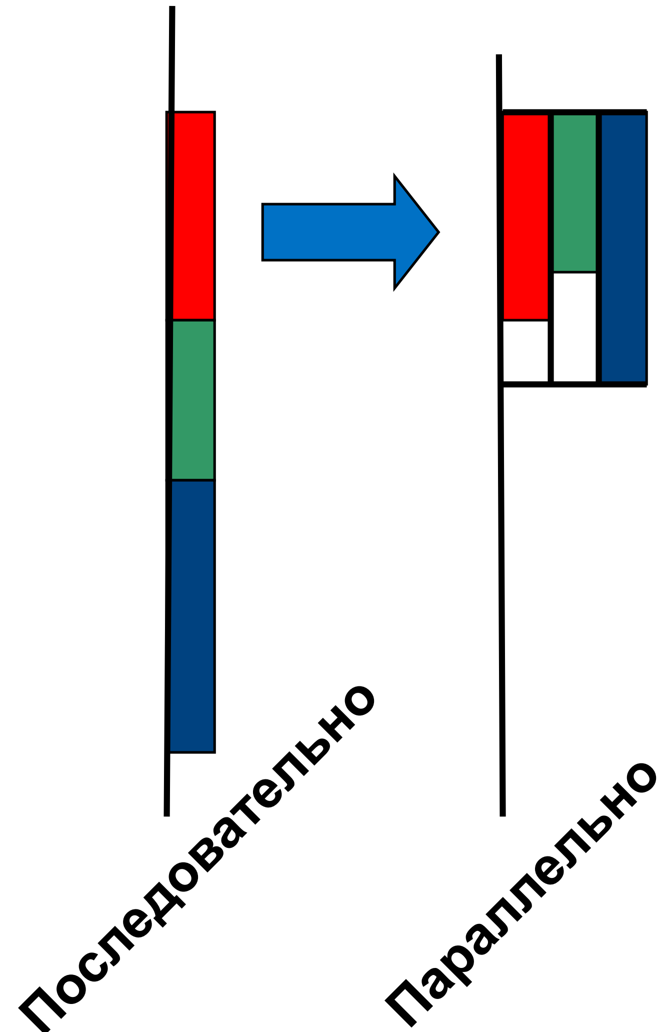
```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- Итерации разделяются блоками по 8
- Если start = 3, то первый блок включает итерации 3, 5, 7, 9, 11, 13, 15, 17

Параллельные секции

Независимые секции кода могут исполняться одновременно

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



Директива Single

Обозначает блок кода, который исполнится только одним потоком

- При этом не специфицируется каким именно

Неявный барьер в конце

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

Директива Master

Обозначает блок кода, который исполнится только основным потоком

Нет неявного барьера в конце

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp master
   {                    // if not master skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

Неявные барьеры

Несколько OpenMP директив имеют неявные барьеры

- **`parallel`**
- **`for`**
- **`single`**

Излишние барьеры отрицательно сказываются на производительности

- Ожидающие потоки не делают полезной работы!

Можно указать OpenMP не использовать неявные барьеры при помощи директивы **`nowait`**

intel

Директива Nowait

```
#pragma omp for nowait
    for(...)
      {...};
```

```
#pragma single nowait
{ [...] }
```

Используйте, когда потоки будут ждать между независимыми вычислениями

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

Директива Barrier

Позволяет задать синхронизацию явно

Каждый поток ждёт пока «не придут» остальные

```
#pragma omp parallel shared (A, B, C)
{
    DoSomeWork(A,B);
    printf("Processed A into B\n");
#pragma omp barrier
    DoSomeWork(B,C);
    printf("Processed B into C\n");
}
```


intel

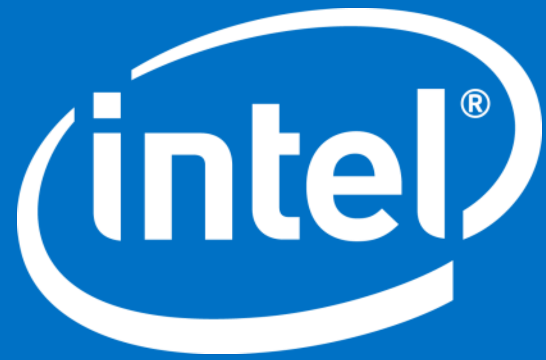
Директива Atomic

Специальный случай критической секции

Применимо только для простых операций записи

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```


intel


intel