Дополнение к занятию 2

Повторение – дополнение

OMP_SCHEDULE (пример - "guided,4" или “dynamic”)

omp_set_schedule()

omp_get_schedule()

Существует понятие совместимости(примечания:DO есть только в языке Fortran, таблица для старой версии стандарта для примера)

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	

Секции

Пример кода:

```
#pragma omp parallel sections
{
#pragma omp section
   {
      Job1();
   }
#pragma omp section
   Job2();

#pragma omp section
   Job3();
}
```

Секции - 2

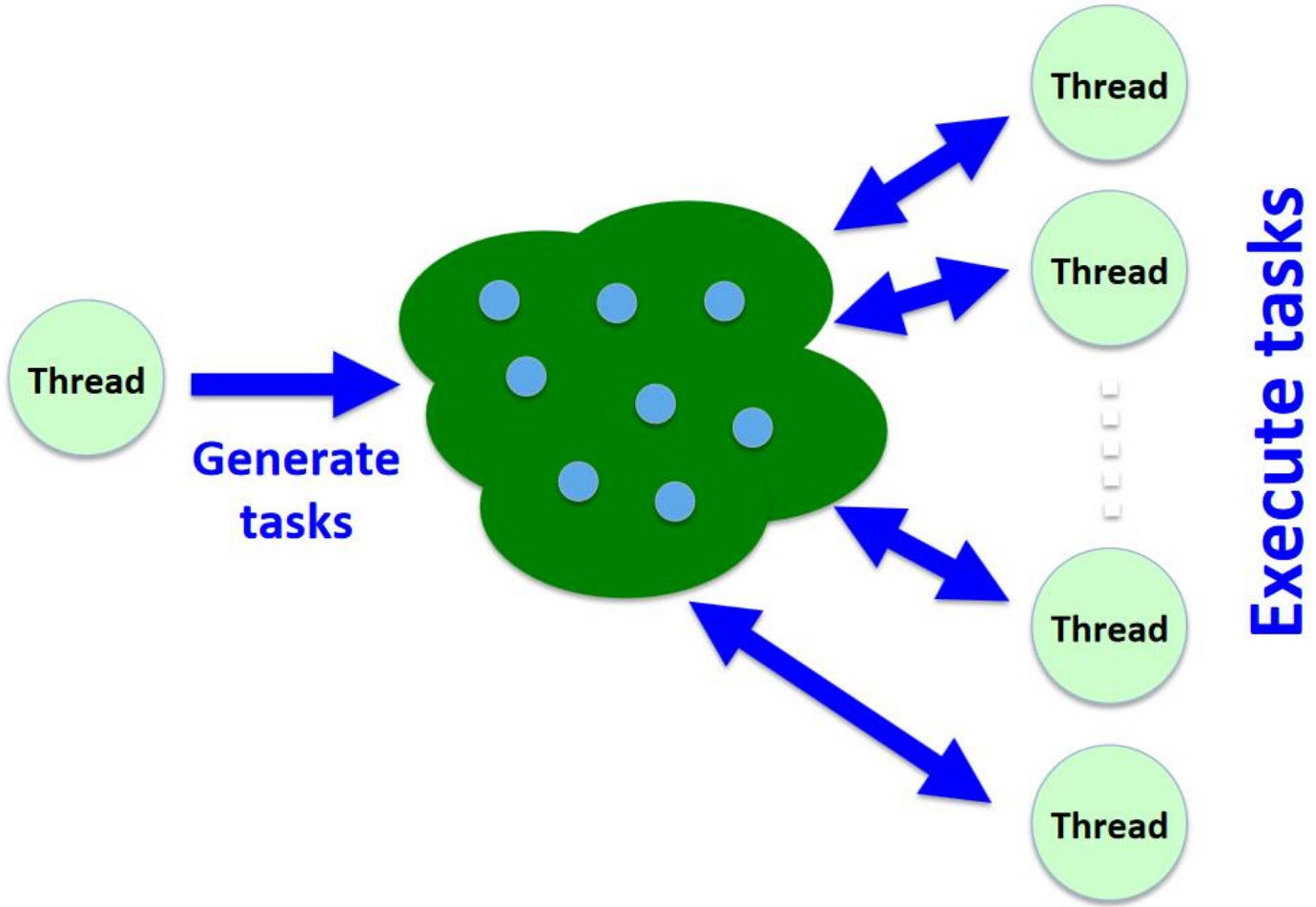
Особенности

1. Синтаксис – требуется директива sections, затем перечисляются секции при помощи отдельной директивы section на каждую секцию, используется только в параллельной области
2. Гарантируется что каждая секция будет выполнена одним из потоков
3. Балансировка – зависит от реализации, не регламентируется
4. Порядок выполнения не регламентирован
5. Присутствует неявная барьерная синхронизация, если не нужна – директива nowait

Задачи (tasks)- основные постулаты

- Блок операторов, который можно выполнить в отдельном потоке, можно вынести в задачу
- При этом новые потоки не создаются, задачи помещаются в специальную область (pool), откуда их может взять любой свободный поток, таким образом производится отход от классической модели «fork/join» с соответствующими накладными расходами
- Появляются возможности для более гибкой работы планировщика и лучшей балансировки ввиду более гибкого распределения потоков между задачами
- Программист должен учитывать условия Бернстайна, чтобы код выполнялся корректно (смотрим лекции за подробностями!)

Задачи (tasks) – схематическое отображение идеи



Задачи (tasks)- особенности реализации

- Начиная с OpenMP 3.0, ряд возможностей – позже
- Поток, встретивший данную конструкцию, может выполнить задачу сразу или поставить её в пул задач
- Задача будет выполнена ровно 1 раз
- Выполнить может не тот поток, который поставил в очередь – а другой из «команды»
- В современных версиях стандарта появилась возможность «привязывать» (tied) или «отвязывать» (untied) задачу, в первом случае она полностью выполняется одним потоком, во втором это не гарантируется (пример – опционально как *)

Задачи (tasks) – синтаксис

- Конструкция task

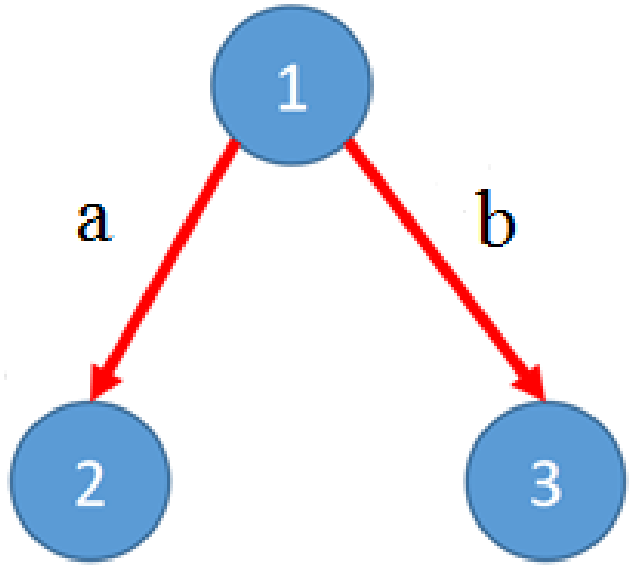
#pragma omp task

Может сочетаться с: //часть только с OpenMP 3.1 или 4.0

- if (выражение) // если выражение не выполняется, выполнение сразу!
- final (выражение)//если выражение, то задание и все его дочерние задачи становятся final и included (т.е. выполняются сразу тем же потоком, разница с if – распространение на иерархию)
- untied //см. предыдущий слайд
- mergeable //может сделаться merged, т.е. с той же средой, что породивший ей процесс
- default (shared | firstprivate | none)
- private (список)
- firstprivate (list)
- shared (список)
- depend (список) // можно явно задавать, от чего зависит выполнение задания)
- priority (значение) // больше – выше приоритет

Пример кода

```
#pragma omp task depend (out a, b)
task1(...);
#pragma omp task depend (in a)
task2(...);
#pragma omp task depend (in b)
task3(….);
```



Директива taskwait и другие

- taskwait - ждём выполнения всех задач, порождённых этим потоком (только один уровень)
- taskyield – можно приостановить эту задачу, если есть другие
- taskgroup – как taskwait, только ждём ещё порождённых дочерними задачами задач, без ограничения на уровень вложенности
- taskloop // и многие другие /////пример!

```
printf(“Before task“);
#pragma omp task
 printf("Task 1 ");
#pragma omp task
 printf("Task 2 ");
#pragma omp taskwait
//какой-то код после выполнения задачи
```

taskloop

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskloop
    for (int i = 0; i < n; i++)
        task_implementation(data[i]);
}
//каждая итерация цикла станет отдельной задачей
```

Средства синхронизации потоков - 1

- single //следующий блок будет выполняться одним потоком, применяется внутри параллельной секции

Пример:

#pragma omp single //внутри параллельной области

#pragma omp task

{} // какое-то задание, которое нужно поставить в пул один раз

- master //следующий блок выполняется мастер-процессом
- barrier //явно описанный барьер, часто ещё встречается неявная барьерная синхронизация

Средства синхронизации потоков - 2

- critical //обозначает критический блок кода – который может в единицу времени выполнять только 1 поток
- atomic //в отличии от критической секции, защищает только одну операцию из ограниченного списка (+, *, -, /, &, ^, |, <<, или >>, не перегруженные (для C++) ++, --, и ряд других конструкций вроде { v = x; x++; })(да, в последних версиях блоки разрешены, но строго из списка), меньше накладные расходы, дополнительные опции read | write | update | capture
- nowait //очень важная директива, если хочется избавиться от скрытой барьерной синхронизации

```
#pragma omp parallel for
for (i = 1; i < 128; i++)
{}
//вот тут скрытый барьер
```

nowait

Особенности:

- не совместима с pragma omp parallel

Т.е.

#pragma omp parallel for nowait = ошибка!

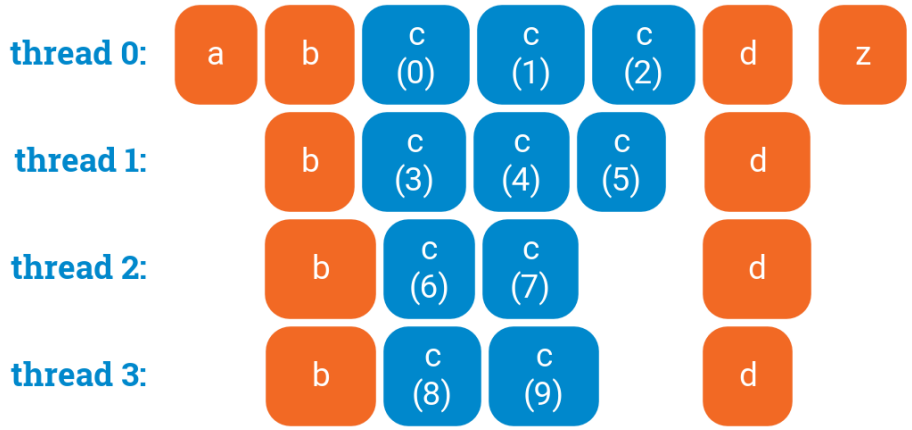
Но!

```
#pragma omp parallel
{
#pragma omp for nowait
for (……..
} – будет работать
```

//совместимость может зависеть от версии стандарта

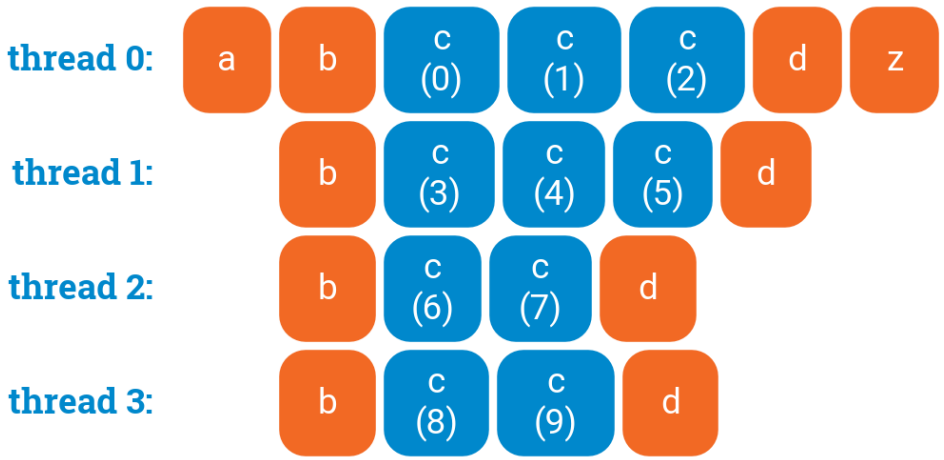
Без nowait

```
a();
#pragma omp parallel
{
   b();
#pragma omp for
    for (int i = 0; i < 10; ++i)
     {
       c(i);
     }
     d();
}
z();
```



C nowait

```
a();
#pragma omp parallel
{
   b();
#pragma omp for nowait
    for (int i = 0; i < 10; ++i)
     {
       c(i);
     }
     d();
}
z();
```



Средства синхронизации потоков - 3

- ordered (применяется для циклов, участок кода, который должен выполняться потоками поочерёдно)

//

Последная «вводная» задача – обеспечить поочерёдный последовательный доступ всех потоков к одной ячейке памяти (например, переменной типа int), действие (например, ++), вывод (номер потока, значение переменной)

- flush - точка, в которой должна быть гарантирована синхронизация содержимого всех видов памяти для всех потоков

Блокировки

Тип данных для простых блокировок - omp_lock_t, схож с мьютексом

Инициализируется функцией void omp_init_lock (omp_lock_t *lock)

После использования - void omp_destroy_lock(omp_lock_t *lock)

Используем:

void omp_set_lock(omp_lock_t *lock) //блокирующая попытка занять lock, если кто-то занял до этого потока – ждём

void omp_unset_lock(omp_lock_t *lock) //при первой возможности

int omp_test_lock(omp_lock_t *lock) //неблокирующая попытка занять //lock

Также есть omp_nest_lock_t, omp_init_nest_lock() и т.д., схож с семафором

Пример кода

```
omp_lock_t simple_lock;
int main() {
  omp_init_lock(&simple_lock);
#pragma omp parallel num_threads(4)
   {
     int tid = omp_get_thread_num();
     while (!omp_test_lock(&simple_lock))
        printf("Thread %d - failed to acquire simple_lock\n", tid);
     printf("Thread %d - acquired simple_lock\n", tid);
     printf("Thread %d - released simple_lock\n", tid);
     omp_unset_lock(&simple_lock);
   }
   omp_destroy_lock(&simple_lock);
}
```