Способы синхронизации данных в OpenMP

Марчевский И.К., Попов А.Ю.

МГТУ им. Н.Э. Баумана

Потребность в синхронизации данных

Программы на OpenMP работают с общей памятью, обращение к которой, если только обработка отдельных элементов данных не происходит нитями абсолютно независимо, требует разрешения конфликтов доступа к памяти. В OpenMP имеется большой набор средств для осуществления синхронизации данных, в т.ч.

- высокоуровневые:
  - критические секции (директива `critical`);
  - atomic-операции (директива `atomic`);
  - барьеры (директива `barrier`);
  - директива `ordered`;
- низкоуровневые:
  - директива `flush`;
  - замки (locks).

Используемая парадигма зачастую называется SPMD (Single Program Multiple Data).

Пример

Задача — суммирование массива длины $n \sim 10^6 \ldots 10^7$ с преобразованием элементов (в демонстрационных целях — для увеличения вычислительной нагрузки).

Для замера времени удобно использовать функцию `omp_get_wtime()` (требует включения заголовочного файла `omp.h`).

Последовательный алгоритм:

```
double sum = 0.0;
for (int i = 0; i < n; ++i)
    sum += f(a[i]);
```

При попытке формально заключить цикл `for` под директиву `#pragma omp parallel` результат будет неверным.

Пример

Задача — суммирование массива длины $n \sim 10^6 \ldots 10^7$ с преобразованием элементов (в демонстрационных целях — для увеличения вычислительной нагрузки).

Для замера времени удобно использовать функцию `omp_get_wtime()` (требует включения заголовочного файла `omp.h`).

Последовательный алгоритм:

```
double sum = 0.0;
for (int i = 0; i < n; ++i)
    sum += f(a[i]);
```

При попытке формально заключить цикл `for` под директиву `#pragma omp parallel` результат будет неверным.
Причина — требуется разделение вычислений по потокам, а также устранение гонки данных (в силу последней вместо p-кратной суммы результат получается меньше).

Параллельный вариант суммирования I

Разделение процесса суммирования: каждая нить суммирует элементы с номерами $\left[id\frac{n}{np}; (id+1)\frac{n}{np}\right)$, либо $id, id+np, id+2np, \ldots$, при этом сохраняя промежуточный результат в элемент с номером id массива сумм. Конфликта при записи данных при этом не происходит. По окончании процесса суммирование локальных результатов производится последовательно (их количество равно количеству нитей, которое обычно невелико по сравнению с размером исходного массива).

Обратить внимание: на время параллельного исполнения влияет способ хранения векторов — массив либо `std::vector`.

Параллельный вариант суммирования II

```
const int np = omp_get_max_threads();
std::vector<double> localSums(np);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for(int i = id; i < n; i += np)
            localSums[id] += f(a[i]);
    }

for(int i = 0; i < np; ++i)
    sum += localSums[i];
```

Критические секции

Критическая секция — фрагмент кода, для которого обеспечивается, что в один момент времени его исполняет только один поток.

- Директива `#pragma omp critical (имя)`
  Имя критической секции указывать необязательно, но оно позволяет более гибко контролировать исполнение секций.
- Следует помнить, что директива `critical` только исключает одновременное исполнение секции кода, но при этом синхронизации потоков нет ни до, ни после нее.
- Следует стараться по возможности реже использовать критические секции, поскольку они снижают степень параллелизма программы.

Atomic-операции

В отличие от универсальных критических секций, для операций чтения/записи/обновления (чтение + запись) данных в памяти имеется аппаратная возможность обеспечения корректности обращения.

- Директива `#pragma omp atomic`, за которой следует только одна строка.
- Блокирование производится только в части обращения к памяти, остальные вычисления выполняются параллельно (например, `sum += f(a[i]);`).
- Обратной стороной повышения производительности является ограниченный набор операций, в C++: инкремент/декремент, арифметические ($+, -, *, /$), побитовые (`&`, `^`, `|`, `<<`, `>>`).
- Начиная с OpenMP 3.1, имеется возможность указать уточняющую опцию `read`/`write`/`update`.

Замки́ (locks) I

Замок — целочисленная переменная (тип данных `omp_lock_t`), которая используется для синхронизации путем вызова функций библиотеки OpenMP.

- Замок может быть в одном из трех состояний: отключен (не инициализирован), разблокирован, заблокирован.
- Замок инициализируется с помощью `omp_init_lock(&lock)` и переходит в разблокированное состояние, с помощью `omp_destroy_lock(&lock)` переводится в неинициализированное состояние.
- Любая нить может «захватить» разблокированный замок с помощью `omp_set_lock(&lock)`, после чего исполнение других нитей по достижении этой команды будет заблокировано. Считывание состояние и «захват» замка производятся с контролем того, что только одна нить может это сделать.
- Освобождение замка производится с помощью `omp_unset_lock(&lock)` только той нитью, которая его «захватила».

Замки́ (locks) II

- Имеется неблокирующая команда попытки «захватить» замок `omp_test_lock(&lock)`, которую необходимо выполнять повторно, если замок заблокирован.

Помимо описанных **простых замков**, имеются **множественные** (nested, тип переменной `omp_nest_lock_t`), состояние «захвата» которых не бинарное, а числовое: «свободен» соответствует 0, «заблокирован» характеризуется положительным числом (повторный захват увеличивает на 1 и может быть выполнен только потоком, изначально «захватившим» замок). Функции — `omp_init_nest_lock(&lock)` и аналогичные.

Начиная с версии OpenMP 4.5, имеется возможность инициализировать замок с подсказкой, например, об интенсивности попыток его заблокировать (большое/небольшое количество нитей одновременно).

Редукция

Для некоторых широко распространенных операций над элементами массивов в библиотеке OpenMP есть средства для корректного их выполнения на программном уровне путем выполнения **редукции**.

- Опция директивы `#pragma omp parallel reduction(op: var)` подразумевает создание локальной копии для каждого потока, в котором будет накапливаться результат обработки его части массива, после чего все копии будут объединены в глобальную переменную для результата `var`.
- Список встроенных операций ограничен арифметическими (`+`, `-`, `*`), побитовыми (`&`, `|`, `^`), логическими (`&&`, `||`) и операциями сравнения (`min`, `max` — в случае C/C++ только с OpenMP 3.1). Локальные переменные инициализируются естественным образом (0 — для `+`, `-`, `|`, `^`, `||`; 1 — для `*`, `&&`; `~0` — для `&`; наименьшее и наибольшее возможные числа — для `max` и `min`).
- Начиная с OpenMP 4.0, имеется возможность объявлять свои операции редукции (удовлетворяющие определенным требованиям) с помощью директивы `declare reduction`.

Параллельные циклы I

При применении директивы `#pragma omp parallel` к циклу вида `for(int i = 0; i < n; ++i)` он будет выполнен всеми потоками целиком. В OpenMP есть возможность запускать цикл на исполнение, указывая, что он должен выполняться потоками совместно (worksharing-loop), при этом разделение итераций между ними выполняется программно и нет необходимости вручную выделять каждому потоку свой объем работы.

- Директива `#pragma omp for` применима только к циклам `for`, причем с жесткими требованиями — заранее известное количество итераций, фиксированный шаг, отсутствие преждевременного выхода из цикла и др. Важно отсутствие переменных, изменяемых по ходу выполнения цикла (loop-carried dependency), т.к. каждый поток начинает исполнение с некой произвольной итерации и не может отследить все изменения.

Параллельные циклы II

- Имеется возможность изменять принцип распределения наборов итераций по потокам с помощью опции `schedule`.
- Можно объединять несколько тесновложенных циклов с помощью опции `collapse(k)` в единое пространство итераций, которое распределяется по потокам.
- Существует объединенная директива `#pragma omp parallel for`. Основные опции (видимость переменных, наличие редукции и пр.) во многом совпадают с `parallel` и `for`.

Пример 2

Задача — поиск максимального элемента массива.
Последовательный алгоритм:

```
double maxVal = -1e100;
double tmp;
for (int i = 0; i < n; ++i) {
    tmp = f(a[i]);
    if (tmp > maxVal)
        maxVal = tmp;
}
```

При $f(x) = x, a_i = n - i$ в случае простейшего алгоритма со сравнением возникает гонка данных — даже при использовании критической секции. Возможно использование редукции (операция `max`), но требуется компилятор, отличный от MS VC++.

Директивы single и master

single

Директива `#pragma omp single` указывает на то, что фрагмент кода будет исполнен только одним потоком (не обязательно с номером 0).

- В конце блока предполагается синхронизация.
- Синхронизация может быть отключена с помощью опции `nowait`.

master

Директива `#pragma omp master` указывает на то, что фрагмент кода будет исполнен потоком с номером 0.

- Синхронизация в конце блока отсутствует. При необходимости ее следует вызывать явно.
- В версии OpenMP 5.1 директива `master` отсутствует, вместо нее доступна более общая директива `masked [filter]` — исполнение фрагмента кода заданным набором потоков (указывается в опции `filter`, при ее отсутствии работает как `master`).

Барьерная синхронизация

Барьерная синхронизация означает, что исполнение дальнейших команд будет производиться только после того, как все нити достигнут барьера.

- Директива `#pragma omp barrier` явно производит барьерную синхронизацию.
- Неявно происходит при выходе из `parallel`, `for`, `single` (без опции `nowait`) и др. конструкций.
- Следует по возможности без явной необходимости не использовать директиву `#pragma omp barrier`, однако она удобна при отладке программы.

Директива `flush`

Поскольку современные вычислительные машины имеют сложную иерархию памяти, в некоторых случаях необходимо явно обеспечивать актуальность данных, общих для различных потоков (при выполнении операций значения в кэше у разных потоков могут отличаться).

- Директива `#pragma omp flush [(переменные)]` обеспечивает согласованность значений указанных переменных (либо всех общих при отсутствии опции, что довольно затратно).
- Вызов `flush` происходит неявно в случае `barrier`, на входе и выходе `parallel`, `critical`, `for` (если не используется `nowait`), при вызове функций для замков, директив `atomic` (только для изменяемой переменной).
- Директива `flush` не обеспечивает синхронизации потоков, а только гарантирует, что потоки будут иметь в памяти самое последнее значение переменных.
- Прирост производительности от использования `flush` обычно не очень велик (доли процентов), но она может иметь значение для корректности работы программы.