Лабораторная работа 3.

Тема: Распараллеливание циклов в OpenMP.

Директива for и ее параметры

Теоретический материал

Директива for

Операторы цикла составляют в среднем 3-5% от объема текста программы, а их исполнение составляет 75-90% времени ее исполнения. Поэтому оптимизация циклов и ускорение вычислений, осуществляемых в них, в том числе и за счет их распараллеливания, является эффективным способом повышения производительности программного обеспечения. В последних версиях компиляторов с включенной опцией компиляции для многоядерного процессора распараллеливание циклов частично осуществляется автоматически. Но в большинстве случаев выполнение этой операции «вручную» позволяет получить лучшие результаты.

Для распараллеливания циклов в Open MP используется директива **for**. Она должна использоваться после или совместно с директивой **parallel** , поскольку не создает параллельной секции, а распределяет работу по выполнению цикла между уже существующими процессами.

```
#pragma omp parallel
... код ...
#pragma omp for опция[[,] опция]…]
<цикл for>
```

Поскольку директивы parallel и for часто следуют друг за другом, допустима их сокращенная запись:

```
#pragma omp parallel for
<цикл for>
```

OpenMP налагает ограничения на циклы for, которые могут быть включены в блок **#pragma omp for** или **#pragma omp parallel for**. Циклы for должны соответствовать следующему формату:

```
for([integer] i = инвариант цикла;
    i {<,>,=,<=,>=} инвариант цикла;
    i {+,-}= инвариант цикла)
```

В OpenMP существуют пять ограничений на то, какие циклы можно распараллелить:

- ✓ Переменная цикла должна иметь тип **signed integer**, беззнаковые целые числа, такие как DWORD, работать не будут;
- ✓ операция сравнения должна иметь следующий формат:

переменная_цикла <,<=,>,>= инвариант_цикла_целого_типа

- ✓ инкрементная часть цикла for должна быть либо целочисленным сложением, либо целочисленным вычитанием и должна практически совпадать со значением инварианта цикла;
- ✓ Если используется операция сравнения < или <=, переменная цикла должна увеличиваться при каждой итерации, а при использовании операции > или >= переменная цикла должна уменьшаться;
- ✓ тело цикла представляет собой структурированный блок, т.е не разрешены переходы из цикла, за исключением оператора exit, который завершает работу всего приложения; если используются операторы goto или break, они должны приводить к переходам внутри цикла, а не вне его. То же самое относится к обработке исключений; исключения должны перехватываться внутри цикла.

Кроме того, при распараллеливании циклов вы должны убедиться в том, что вычисление текущей итерации цикла не зависит от результатов вычисления предыдущих итераций. В этом случае программа может выполнять цикл в любом порядке, в том числе и параллельно. Соблюдение этого важного требования компилятор не проверяет, это задача программиста. Попытка распараллелить цикл, содержащий информационные зависимости между итерациями, не приведет к сообщению об ошибке, и программа будет успешно откомпилирована и (возможно) даже выполнена, но результаты вычислений будут ошибочны.

Если необходимо распараллелить цикл, содержащий подобную информационную зависимость, необходимо определить, есть ли возможность от нее избавиться, например, заменив итерационные вычисления явной формулой.

Например, в цикле выполняются следующие вычисления:

```
a[0]=x;
a[i]=a[i-1]+x, i=1..N
```

Здесь присутствует информационная зависимость между итерациями: чтобы вычислить a[i], нужно знать значение a[i-1]. Если одному потоку поручить вычислить значения элементов от 1 до N/2, а другому параллельно – от N/2+1 до N, то результаты работы второго потока будут неверны.

В данном случае итерационные вычисления можно заменить вычислением выражения:

```
a[i]=x*(i+1), i=0..N
```

Данное выражение не содержит зависимостей, и вычисления могут выполняться в произвольном порядке.

Если невозможно избавиться от зависимостей, можно использовать вместо параллелизма по данным функциональный, либо синхронизировать потоки более сложным способом.

Переменная цикла, который следует за директивой for, автоматически определяется как private (локальная), т. е. в каждом потоке используется свой экземпляр перемен-

ной. Для прочих переменных категории private (локальная) и shared (разделяемая) должны быть определены явно при необходимости.

Планирование исполнения цикла

Как уже было указано, директива for распределяет работу по выполнению цикла между отдельными процессами. Выполнить это распределение можно различными способами. Как именно выполняется это распределение, определяется по умолчанию либо с помощью опции **schedule**

#pragma omp parallel for schedule (алгоритм планирования [, параметр – число итераций])

Статическое планирование

Используется по умолчанию или может быть указано явно при использовании опции schedule.

schedule(static[, число_итераций])

Блочно-циклическое распределение итераций цикла; размер блока – определяет параметр «число итераций».

Первый блок итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение параметра не указано, то если n – число итераций цикла, а T – число потоков в секции, каждый поток выполнит n/T итераций. Точное распределение потоков в том случае, когда результат деления не целочисленный, зависит от реализации Open MP, но в любом случае гарантируется исполнение всех итераций цикла.

Примеры

```
#pragma omp parallel for schedule(static, 10)
 for(int i = 1; i <= 101; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

Поток 0 получает право на выполнение 1-10, 41-50, 81-90

Поток 1 получает право на выполнение итераций 11-20, 51-60, 91-100

Поток 2 получает право на выполнение итераций 21-30, 61-70, 101

Поток 3 получает право на выполнение итераций 31-40, 71-80

```
#pragma omp parallel for
 for(int i = 1; i <= 101; i++)
```

или

```
#pragma omp parallel for schedule(static)
 for(int i = 1; i <= 101; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

Поток 0 получает право на выполнение 1-26

Поток 1 получает право на выполнение итераций 27-52

Поток 2 получает право на выполнение итераций 53-78

Поток 3 получает право на выполнение итераций 79-101

Динамическое планирование

```
schedule(dynamic[, число_итераций])
```

Динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает число итераций, соответствующее параметру опции.

По умолчанию число итераций равно 1.

Та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из того же размера из оставшихся итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

Примеры

```
#pragma omp parallel for schedule(dynamic, 15)
 for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

Поток 0 получает право на выполнение итераций 1-15.

Поток 1 получает право на выполнение итераций 16-30.

Поток 2 получает право на выполнение итераций 31-45.

Поток 3 получает право на выполнение итераций 46-60.

Поток 3 завершает выполнение итераций.

Поток 3 получает право на выполнение итераций 61-75.

Поток 2 завершает выполнение итераций.

Поток 2 получает право на выполнение итераций 76-90.

Поток 0 завершает выполнение итераций.

Поток 0 получает право на выполнение итераций 91-100.

Управляемое планирование

schedule(guided[, число_итераций])

Динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины, определяемой параметром опции (по умолчанию параметр=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл.

Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения параметра.

число выполняемых потоком итераций =
max (число нераспределенных итераций / omp_get_num_threads(), число итераций)

Примеры

```
#pragma omp parallel for schedule (guided, 10)
 for (int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

Поток 0 получает право на выполнение итераций 1-25.

Число оставшихся итераций: 100-25=75. Max (75/4~19, 10)=19

Поток 1 получает право на выполнение итераций 26-44.

Число оставшихся итераций: 75-19=56. Max (56/4=14, 10)=14

Поток 2 получает право на выполнение итераций 45-59.

Число оставшихся итераций: 56-14=42. Max (42/4~10, 10)=10.

С этого момента и до конца цикла все потоки будут получать по 10 итераций.

Поток 3 получает право на выполнение итераций 60-69.

Поток 3 завершает выполнение итераций.

Поток 3 получает право на выполнение итераций 70-79.

Поток 2 завершает выполнение итераций.

Поток 2 получает право на выполнение итераций 80-89.

Поток 3 завершает выполнение итераций.

Поток 3 получает право на выполнение итераций 90-99.

Поток 1 завершает выполнение итераций.

Поток 1 получает право на выполнение 100 итерации.

Планирование в период выполнения

schedule(runtime)

Способ распределения итераций выбирается во время работы программы по значению переменной среды **OMP_SCHEDULE**. Изменить значение переменной можно с помощью функции **omp_set_schedule.** По умолчанию значение переменной соответствует статическому способу планирования выполнения цикла.

Автоматическое планирование (OpenMP 3.0).

schedule(auto)

Способ распределения итераций выбирается компилятором и/или системой выполнения.

Управление последовательностью выполнения итераций

В общем случае порядок выполнения итераций цикла при распараллеливании может быть произвольным. Это позволяет оптимизировать загрузку процессора. Но в некоторых случаях есть необходимость часть кода из тела цикла выполнить в порядке, соответствующем последовательному выполнению цикла.

Для обеспечения этой возможности используется комбинация опции и директивы с общим именем **ordered.**

Указание в директиве for опции ordered сигнализирует о том, что в теле цикла, соответствующего директиве, будет присутствовать директива ordered. Разрешено использовать только один ordered блок на цикл.

```
#pragma omp parallel
... код ...
#pragma omp for ordered
<заголовок цикла>
{
... тело цикла...
#pragma omp ordered
{
... код, выполняемый в порядке последовательного выполнения цикла ...
}
... тело цикла...
}
```

Пример

```
#pragma omp parallel
#pragma omp for ordered schedule(dynamic)
 for(int n=0; n<100; ++n)
 {
   files[n].compress();
   #pragma omp ordered
   send(files[n]);
 }
```

Цикл выполняет вызов и исполнение функции **compress** в параллельном режиме и произвольном порядке, но выполнение функции **send** выполняется в последовательном порядке. Если, например, поток «сжал» седьмой файл, но шестой файл к этому моменту ещё не был «отправлен», поток будет ожидать «отправки» шестого файла. Каждый файл «сжимается» и «посылается» один раз, но «сжатие» может происходить в параллельном режиме.

Другие опции директивы for

Опции, управляющие работой с переменными

private (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

firstprivate (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

lastprivate (список) – переменным, перечисленным в списке, присваивается результат, полученный в последней секции;

reduction (оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после завершения всех секций выполняется заданный оператор; оператор это: +, *, -, &, |, ^, &&, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;

nowait – в конце блока секций происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет нитям, уже дошедшим до конца своих секций, продолжить выполнение без синхронизации с остальными.

Описание лабораторной работы 3

Порядок выполнения

1. Ознакомиться с описанием лабораторной работы. Вариант задания совпадает с вариантом задания на лабораторную работу 1.
2. Проанализировать программу ЛР1 и определить, какие фрагменты программы целесообразно/возможно распараллелить, используя директиву **for**
3. Распараллелить программу с использованием директивы **#pragma omp for** (статическое планирование исполнения)
4. Убедиться, что прежняя и новая версии выдают одинаковые результаты.
5. Зафиксировать время выполнения программы при различных размерах обрабатываемых массивов.
6. Изменить способ планирования выполнения циклов на любой другой. Повторить эксперимент.

Содержание отчета

Полный отчет предоставляется в электронном виде и/или твердой копии (по требованию преподавателя). Он должен содержать:

1. Титульный лист.
2. Описание полученного задания;
3. Фрагмент кода, в котором проведено распараллеливание.
4. Скриншоты, демонстрирующие результат работы программы и показатели загрузки системы (используется диспетчер задач или его аналог);
5. Таблица из отчетов к ЛР1,2, дополненная результатами тестирования времени исполнения данной программы.
6. Выводы (ОБЯЗАТЕЛЬНО!).