Параллельное программирование на OpenMP

Оглавление

Введение

Параллельное программирование

Параллельное программирование применяется тогда, когда для последовательной программы требуется уменьшить время ее выполнения, или когда последовательна программа, в виду большого объема данных, перестает помещаться в память одного компьютера. Направление развития в области высокопроизводительных вычислений как раз направлено на решение этих двух задач: создание мощных вычислительных комплексов с большим объемом оперативной памяти с одной стороны и разработка соответствующего ПО с другой.

По сути весь вопрос заключается в минимизации соотношения цена/производительность. Ведь всегда можно построить (собрать) вычислительную систему, которая будет эффективно решать поставленную задачу, но адекватна ли будет при этом цена такого решения. Можно выделить два направления развития компьютерной техники: векторные машины (Cray) и кластеры (обычные компьютеры, стандартное ПО).

Написание параллельных программ

Разработка параллельных программ (ПП) состоит из трех основных этапов:

Декомпозиция задачи на подзадачи. Идеально, чтобы эти подзадачи работали независимо друг от друга (принцип локальности данных). Обмен данными между подзадачами является дорогой операцией, особенно, если это обмен по сети.

Распределение задачи по процессорам (виртуальным процессорам). В некоторых случаях решение этого вопроса можно оставить на усмотрение среды выполнения ПП.

Написание программы с использованием какой-либо параллельной библиотеки. Выбор библиотеки может зависеть от платформы, на которой программа будет выполняться, от требуемого уровня производительности и от природы самой задачи.

Параллельные архитектуры

В массе своей все вычислительные комплексы и компьютеры делятся на три группы:

Системы с распределенной памятью. Каждый процессор имеет свою память и не может напрямую доступаться к памяти другого процессора.

Разрабатывая программы подобные системы программист в явном виде должен задать всю систему коммуникаци (Передача сообщений – Message Passing). Библиотеки: MPI, PVM, Shmem (Cray only).

Системы с общей (разделяемой) памятью. Процессор может напрямую обращаться в память другого процессора. Процессоры могут сидеть на одной шине (SMP). Разделяемая память может быть физически распределенной, но тогда стоимость доступа к удаленной памяти может быть очень высока и это должен учитывать разработчик ПП.

Подходы к разработке ПО: Threads, директивы компилятора (OpenMP), механизм передачи сообщения.

Комбинированные системы. В кластерах могут объединяться компьютеры различной конфигурации.

OpenMP

Введение в OpenMP

OpenMP – механизм написания параллельных программ для систем с общей памятью.

Состоит из набора директив компилятора и библиотечных функций.

Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran.

Поддерживается производителями аппаратуры (Intel, HP, SGI, Sun, IBM), разработчиками компиляторов (Intel, Microsoft, KAI, PGI, PSR, APR, Absoft)

Программная модель OpenMP

Основной поток порождает дочерние потоки по мере необходимости.

Модель fork-join.

Программирование путем вставки директив компилятора в ключевые места исходного кода программы.

Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для расспараллеливания участков кода.

Последовательный код	Параллельный код

`void main(){`
`  double x[1000];`
`  for(i=0; i<1000; i++){`
`  calc_smth(&x[i]);`
`  }`
`}`
	`void main(){`
	`  double x[1000];`
	`#pragma omp parallel for ...`
	`  for(i=0; i<1000; i++){`
	`  calc_smth(&x[i]);`
	`  }`
	`}`

Директива **`#pragma omp parallel for`** указывает на то, что данный цикл следует разделить по итерациям между потоками.

Количество потоков можно контролировать из программы, или через среду выполнения программы- переменную окружения **`OMP_NUM_THREADS`**.

Следует отметить, что разработчик ответственен за синхронизацию потоков и зависимость между данными.

Для того, чтобы скомпилировать программу с поддержкой OpenMP компилятору следует указать дополнительный ключ:

```
icc -openmp prog.c
ifc -openmp prog.f
```

Как взаимодействуют потоки?

В модели с разделяемой памятью взаимодействие потоков происходит через разделяемые переменные. При неакуратном обращении с такими переменными в программе могут возникнуть ошибки соревнования (race condition). Такое происходит из за того, что потоки выполняются параллельно и соответственно последовательность доступа к разделяемым переменным может быть различна от одного запуска программы к другому.

Для контроля ошибок соревнования работу потоков необходимо синхронизировать. Для этого используются такие примитивы синхронизации как критические секции, барьеры, атомарные операции и блокировки[1]. Стоит отметить, что синхронизация может потребовать от программы дополнительных накладных расходов и лучше подумать и распределить данные таким образом, чтобы количество точек синхронизации было минимизировано.

1 В литературе также встречается термин «замок» (от англ. lock)

Основы OpenMP

Синтаксис

В основном конструкции OpenMP – это директивы компилятора

Для C/C++ директивы имеют следующий вид:

```
#pragma omp конструкция [условие [условие]...]
```

Для Fortran-а директивы принимают одну из следующих форм:

```
c$omp конструкция [условие [условие]...]
!$omp конструкция [условие [условие]...]
*$omp конструкция [условие [условие]...]
```

Поскольку конструкции OpenMP являются директивами, то тот компилятор, который их не понимает, пропустит их и все же соберет OpenMP программу, правда последовательную.

В большинстве своем директивы OpenMP применимы только к структурным блокам, которые имеют единственную точку входа и единственную точку выхода. Единственным исключением является оператор STOP в языке Fortran и функция exit() в C/C++.

Правильно

```
#pragma omp parallel
{
L1:
  wrk[id] = junk[id];
  res[id] = wrk[id]*1.342;
  if(conv(res)) goto L1;
}
  printf("%d", id);
```

Неправильно

```
#pragma omp parallel
{
L1:
  wrk[id] = junk[id];
L2:
  res[id] = wrk[id]*1.342;
  if(conv(res)) goto L3;
  goto L1;
}
  if(not_done) goto L2;
L3:
  printf("%d", id);
```

Параллельные регионы

Параллельные регионы являются основным понятием в OpenMP. Именно там, где задан этот регион программа исполняется параллельно. Как только компилятор встречает прагму omp parallel, он вставляет инструкции для создания параллельных потоков.

Выше уже упоминалось, что количество порождаемых потоков для параллельных областей

контролируется через переменную окружения OMP_NUM_THREADS, а также может задаваться через вызов функции внутри программы.

Каждый порожденный поток исполняет блок код в структурном блоке. По умолчанию синхронизация между потоками отсутствует и поэтому последовательность выполнения конкретного оператора различными потоками не определена.

После выполнения параллельного участка кода все потоки, кроме основного завершаются, и только основной поток продолжает исполняться, но уже один.

Каждый поток имеет свой уникальный номер, который изменяется от 0 (для основного потока) до количества потоков – 1. Идентификатор потока может быть определен с помощью функции omp_get_thread_num().

Зная идентификатор потока, можно внутри области параллельного исполнения направит потоки по разным ветвям.

```
#pragma omp  parallel
{
   myid = omp_get_thread_num();
   if(myid == 0)
     do_something();
   else
     do_something_else(myid);
}
```

Приведенный пример обладает одним недостатком. Каким? (*мы не знаем является ли переменна myid разделяемой или приватной*).

Область параллельного исполнения описывается в C/C++ следующим образом:

```
#pragma omp  parallel \
 shared(var1, var2, ....)  \
 private(var1, var2, ...)  \
 firstprivate(var1, var2, ...)  \
 reduction(оператор:var1, var2, ...)  \
 if(выражение)  \
 default(shared|none)
{
    структурный блок
}
```

На Фортране:

```
c$omp  parallel
c$omp& shared(var1, var2, ....)
c$omp& private(var1, var2, ...)
c$omp& firstprivate(var1, var2, ...)
c$omp& reduction(оператор:var1, var2, ...)
c$omp& if(выражение)
c$omp& default(private|shared|none)

    структурный блок

c$omp end parallel
```

Модель исполнения

Существует две модели исполнения: динамическая, когда количество используемых потоков в программе может варьироваться от одной области параллельного выполнения к другой, и статическая, когда количество потоков фиксировано.

Модель исполнения контролируется или через переменную окружения OPM_DYNAMIC или с помощью вызова функции omp_set_dynamic().

Конструкции OpenMP

Условия выполнения

Условия выполнения определяют то, как будет выполняться параллельный участок кода и область видимости переменных внутри этого участка кода. Опишем следующие условия:

`shared(var1, var2, ....)`
Условие shared указывает на то, что все перечисленные переменные будут разделяться между потоками. Все потоки будут доступаться к одной и той же области памяти.

`private(var1, var2, ...)`
Условие private указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

`firstprivate(var1, var2, ...)`
Это условие аналогично условию `private` за тем исключением, что указанные переменные инициализируются при входе в параллельный участок кода значением, которое имела переменна до входа в параллельную секцию.

`lastprivate(var1, var2, ...)`
Приватные переменные сохраняют свое значение, которое они получили при достижении конца параллельного участка кода.

`reduction(оператор:var1, var2, ...)`
Это условие гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.

`if(выражение`)
Это условие говорит о том, что параллельное выполнение необходимо только если выражение истино.

default(shared|private|none)
Это условие определяет область видимости переменных внутри параллельного участка кода по умолчанию.

schedule(type[,chank])
Этим условием контролируется то, как итерации цикла распределяются между потоками.

Условия private, shared, default

Рассмотрим следующие примеры:

```
#pragma omp parallel shared(a) private(myid, x)
{
   myid = omp_get_thread_num();
   x = work(myid);
   if(x < 1.0)
     a[myid] = x;
}
```

```
#pragma omp parallel default(private) shared(a)
{
   myid = omp_get_thread_num();
   x = work(myid);
   if(x < 1.0)
     a[myid] = x;
}
```

В обоих примерах каждый поток будет иметь свою копию переменных **x** и **myid**. Если эти переменные не будут объявлены как приватные, то их значение во врем выполнения будет не определено. Значение переменных **x** и **myid** при входе в параллельный участок кода не определено и требуется инициализация этих переменных.

Во втором примере условие **default** автоматически указывает компилятору на необходимость завести свои переменные **x** и **myid** для каждого потока.

Условие **shared** в примерах говорит о том, что массив **a** является разделяемым между потоками и его значение сохраняется при выходе из параллельного участка кода.

Условие firstprivate

Переменные, которые подпадают под это условие являются приватными для каждого потока, но перед выполнением потока происходит их инициализация значением, которое было получено в предыдущем последовательном коде. Так в следующем примере до входа в параллельный участок кода значение переменной **a** равнялось 10. Это же значение имеет эта переменна и при входе в параллельный участок кода.

```
int myid, a;

a = 10;
#pragma omp parallel default(private) \
                     firstprivate(a)
{
   myid = omp_get_thread_num();
   printf("Thread%d: a = %d\n", myid, a);
   a = myid;
   printf("Thread%d: a = %d\n", myid, a);
}
```

Вывод

```
Thread1: a = 10
Thread1: a = 1
Thread2: a = 10
Thread0: a = 10
Thread3: a = 10
Thread3: a = 3
Thread2: a = 2
Thread0: a = 0
```

Конструкции OpenMP для распределения работ

Использование других условий более наглядно будет продемонстрировано на примере конструкций разделения работ. Таких конструкций всего три:

- параллельный цикл **for/DO**
- параллельные секции (**sections**)
- Конструкция **single**

Параллельный цикл **for/DO**

Цель конструкции – распределение итераций цикла по потокам.

```
#pragma omp parallel
{
#pragma omp for private(i) shared(a,b)
  for(i=0; i<10000; i++)
    a[i] = a[i] + b[i]
}
```

По умолчанию барьером для потоков является конец цикла. Все потоки достигнув конца цикла дожидаются тех, кто еще не завершился, после чего основная нить продолжает выполняться дальше. Используя условие **nowait** для цикла можно разрешить основной нити не дожидаться завершения дочерних нитей.

Директива параллельного цикла for/DO имеет следующий синтаксис:

Для C/C++

```
#pragma omp for [условие [,условие] ...]
 цикл for
```

где условие – это одно из:

```
private(var1, var2, ...)
shared(var1, var2, ...)
firstprivate(var1, var2, ...)
lastprivate(var1, var2, ...)
reduction(оператор: var1, var2, ....)
ordered
schedule(тип [, размер блока])
nowait
if(выражение)
```

Для Фортрана

```
c$omp do [условие [,условие] ...]
 цикл DO
[c$omp end do [nowait]]
```

где условие – это одно из:

```
private(var1, var2, ...)
shared(var1, var2, ...)
firstprivate(var1, var2, ...)
lastprivate(var1, var2, ...)
reduction(оператор: var1, var2, ....)
ordered
schedule(тип [, размер блока])
if(выражение)
```

Параллельные секции

Порой возникает необходимость параллельно выполнить действия, которые не являются итерациями цикла. Конечно можно воспользоваться для этих целей простой директивой parallel, но тогда придется писать дополнительный код, чтобы различную работу распределить между потоками. Более просто эту задачу можно решить с помощью параллельных секций.

```
#pragma omp parallel sections
{
#pragma omp section
  {
  printf("T%d: foo\n", omp_get_thread_num());
  }
#pragma omp section
  {
  printf("T%d: bar\n", omp_get_thread_num());
  }
}// omp sections
```

Каждая секция выполняется в отдельном потоке, что позволяет производить декомпозицию по коду. Точкой синхронизации является конец блока **sections**. В случае, когда необходимо чтобы основной поток не ждал завершения остальных потоков следует использовать условие **nowait**.

Синтаксис параллельных секций в C/C++

```
#pragma omp sections \
           [условие [,условие...]]
{
    #pragma omp section
        структурный блок
   [#pragma omp section
        структурный блок
    ...]
}
```

где условие – это одно из:

```
private(var1, var2, ...)
firstprivate(var1, var2, ...)
lastprivate(var1, var2, ...)
reduction(оператор: var1, var2, ....)
nowait
```

Синтаксис параллельных секций Fortran

```
c$omp sections [условие [,условие...]]
c$omp section
        структурный блок
[c$omp section
        структурный блок
    ...]
c$omp end sections [nowait]
```

где условие – это одно из:

```
private(var1, var2, ...)
firstprivate(var1, var2, ...)
lastprivate(var1, var2, ...)
reduction(оператор: var1, var2, ....)
```

Конструкция **`single`**

Если в параллельной секции требуется выполнить какое-либо действие и при этом это действие должно быть выполнено только одним потоком (например, подсчет промежуточного результата), то для этого идеально подходит конструкция **`single`**.

Синтаксис в C/C++

```
#pragma omp single [условие [, условие ...]]
    структурный блок
```

где условие – это одно из:

```
private(var1, var2, ...)
firstprivate(var1, var2, ...)
nowait
```

Синтаксис в Fortran

```
c$omp single [условие [, условие ...]]
    структурный блок
c$omp end single [nowait]
```

где условие – это одно из:

```
private(var1, var2, ...)
firstprivate(var1, var2, ...)
```

Условия выполнения (2)

Условие `if`

В тех случаях, когда накладные расходы на порождение потоков могут быть больше, чем выигрыш от распараллеливания, то необходимо воспользоваться условием **`if`**.

```
#pragma omp parallel
{
#pragma omp for if(n>2000)
  {
    for(i=0; i<n; i++)
      a[i] = work(i);
  }
}
```

В приведенном примере цикл будет расспараллелен при условии, что итераций цикла больше, чем 2000.

Условие `lastprivate`

Это условие действует аналогично условию private за тем исключением, что значение переменной, вычисленное на последней итерации цикла, сохраняется.

```
#pragma omp parallel
{
#pragma omp for private(i) lastprivate(k)
  for(i=0; i<10; i++)
      k = i*i;
}
  printf("k = %d\n", k);
```

При выходе из цикла значение переменной **`k`** будет равно 100. Если бы переменная **`k`** была объявлена как приватная, то ее значение при выходе из цикла будет неопределено.

Условие `reduction`

Это условие позволяет производить безопасное глобальное вычисление. Приватная копия каждой перечисленной переменной инициализируется при входе в параллельную секцию в соответствии с указанным оператором (0 для оператора +). При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее и передается в основной поток.

```
#pragma omp parallel
{
#pragma for shared(x) private(i) reduction(+:sum)
    for(i=0; i<10000; i++)
          sum += x[i];
}
```

```
#pragma omp parallel
{
#pragma for shared(x) private(i) reduction(min:gsum)
    for(i=0; i<10000; i++)
          gmin = min(gmin, x[i]);
}
```

Доступны следующие операторы и агрегатные функции:

В C/C++

```
+, - , *, &, ^, |, &&, ||
min, max
```

В Фортране

```
+, -, *, .and., .or., .eqv., .neqv.
min, max, iand, ior, ieor
```

Условие `schedule`

Данное условие контролирует то, как работа будет распределяться между потоками.

```
schedule(тип [, размер блока])
```

Размер блока задает размер каждого пакета на обработку потоком (количество итераций). Тип расписания может принимать следующие значение:

- `static` – итерации равномерно распределяются по потокам. Т.е. если в цикле 1000 итераций и 4 потока, то один поток обрабатывает все итерации с 1 по 250, второй – с 251 по 500, третий - с 501 по 750, четвертый с 751 по 1000. Если при этом задан еще и размер блока, то все итерации блоками заданного размера циклически распределяются между потоками.

  Статическое распределение работы эффективно, когда время выполнения итераций равно, или приблизительно равно. Если это не так, то разумно использовать следующий тип распределения работ.

- `dynamic` – работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками. Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую.

  Стоит отметить, что при этом подходе несколько большие накладные расходы, но можно добиться лучшей балансировки загрузки между потоками.

- `guided` – данный тип распределения работы аналогичен предыдущему, за тем исключением, сто размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения.

При таком подходе можно достичь хорошей балансировки при меньших накладных расходах.

- `runtime` – тип распределения определяется в момент выполнения программы. Это удобно в экспериментальных целях для выбора оптимального значения **типа** и **размера блока**.

  Тип распределения работ зависит от переменной окружения **`OMP_SCHEDULE`**. По умолчанию считается, что установлен статический метод распределения работ.

```
bash> export OMP_SCHEDULE=static,1000
bash> export OMP_SCHEDULE=dynamic
```

Условие `ordered`

Переменные окружения OpenMP

OMP_NUM_THREADS
: Устанавливает количество потоков в параллельном блоке. По умолчанию, количество потоков равно количеству виртуальных процессоров.

OMP_SCHEDULE
: Устанавливает тип распределения работ в параллельных циклах с типом runtime.

OMP_DYNAMIC
: Разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации.

OMP_NESTED
: Разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию – запрещено.

Библиотечные функции OpenMP

Для эффективного использования процессорного времени компьютера и написания гибких OpenMP программ пользователю предоставляется возможность управлять ходом выполнения программы по средством библиотечных функций. Библиотека OpenMP предоставляет пользователю следующий набор функций:

`void omp_set_num_threads(int num_threads)`
: Устанавливает количество потоков, которое может быть запрошено для параллельного блока.

`int omp_get_num_threads()`
: Возвращает количество потоков в текущей команде параллельных потоков.

`int omp_get_max_threads()`
: Возвращает максимальное количество потоков, которое может быть установлено omp_set_num_threads.

int omp_get_thread_num()
Возвращает номер потока в команде (целое число от 0 до количества потоков – 1).

int omp_get_num_procs()
Возвращает количество физических процессоров доступных программе.

int omp_in_parallel()
Возвращает не нулевое значение, если вызвана внутри параллельного блока. В противном случае возвращается 0.

void omp_set_dyamic(expr)
Разрешает/запрещает динамическое выделение потоков.

int omp_get_dynamic()
Возвращает разрешено или запрещено динамическое выделение потоков.

void omp_set_nested(expr)
Разрешает/запрещает вложенный параллелизм.

int omp_get_nested()
Возвращает разрешен или запрещен вложенный параллелизм.

Перед использованием функций в фортране следует из объявить как соответствующий тип данных, в C/C++ - подключить файл заголовков omp.h.

```
#include <omp.h>
```

Изменения, сделанные функциями, являются приоритетнее, чем соответствующие переменные окружения. Так, функция `omp_set_num_threads()` переписывает значение переменной окружения **`OMP_NUM_THREADS`**, которое может быть установлено перед запуском программы.

Зависимость по данным

Для того, чтобы цикл мог быть распараллелен, работа, которая выполняется на одной итерации цикла не должна зависеть от от работы на другой итерации. Другими словами, итерации цикла должны быть независимыми. Порой от зависимости по данным можно избавиться слегка переписав код.

`for(i=1; i<8; i++)` `  a[i] = c*a[i-1];`	`Зависимость есть`
`for(i=1; i<9; i+=2)` `  a[i] = c*a[i-1];`	`Зависимости нет`

Утверждение 1

Только те переменные, в которые происходит запись на одной итерации и чтение их значения на другой создают зависимость по данным.

Утверждение 2

Только разделяемые переменные могут создавать зависимость по данным.

Следствие. Если переменная не объявлена как приватная, она может оказаться разделяемой

и привести к зависимости по данным.

```
for(i=0; i<1000; i++){
    x = cos(a[i]);
    b[i] = sqrt(x*c);
}
```

Вызовы функций внутри циклов – обычное дело. Однако и такие циклы могут быть распараллелены. Для этого программист должен сделать функцию независимой от внешних данных кроме как от **значения** параметров. В функции так же не должно быть статических переменных (**`static`**).

```
double foo(double *a, double *b, int i){
// Зависимость есть
...
 return 0.345*(a[i] + b[2*i]*C);
}

double bar(double a, double b){
// Зависимости нет
 return 0.345*(a + b*C);
}
```

Иногда возникают ситуации когда индексы одного массива приходится хранить в другом массиве.

```
for(i=0; i<N; i++){
    b[i] = c*a[indx1[i]];
}

for(i=0; i<N; i++){
    b[indx2[i]] = sqrt(a[i]);
}
```

В приведенном выше примере если indx1[i] не равен i на каждой итерации, то есть зависимость по данным. Если в массиве indx2 есть повторения, то в цикле есть зависимость итераций по данным.

Циклы, в которых есть выход по условию не должны подвергаться распараллеливанию, поскольку эти циклы требуют упорядоченного выполнения.

```
for(i=0; i<1000; i++){
    b[i] = sqrt(cos(a[i])*c);
    if(b[i]>epsilon)
        break;
}
```

Рассмотрим еще один пример:

```
for(k=0; k<N; k++)
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            a[i][j] += b[i][k]*c[k,j];
```

Если внешний цикл распараллелить, то получается зависимость по данным – `a[i][j]`. Для

исправления такого положения вещей цикл по `k` следует сделать внутренним.

```
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        for(k=0; k<N; k++)
            a[i][j] += b[i][k]*c[k,j];
```

Порой бывает трудно определить есть ли зависимость по данным в коде и тогда на помощь разработчику может прийти компилятор.

Средства синхронизации в OpenMP

В OpenMP предусмотрены следующие конструкции синхронизации:

`critical` – критическая секция

`atomic` – атомарность операции

`barrier` – точка синхронизации

`master` – блок, который будет выполнен только основным потоком. Все остальные потоки пропустят этот блок. В конце блока неявной синхронизации нет.

`ordered` – выполнять блок в заданной последовательности

`flush` – немедленный сброс значений разделяемых переменных в память.

Критическая секция

Наличие критической секции в параллельном блоке гарантирует, что она в каждый конкретный момент времени будет выполняться только одним потоком. Т.е. когда один поток находится в критической секции, все остальные потоки, которые готовы в нее войти, находятся в приостановленном состояние. Критические секции могут снабжаться именами. При этом критические секции считаются независимыми, только если они используют разные имена. По умолчанию, все непроименованые критические секции имеют одно имя.

Синтаксис критической секции на C/C++

```
#pragma omp critical [(имя)]
    Структурный блок
```

На Фортране

```
c$omp critical [(имя)]
    Структурный блок
c$omp end critical
```

Пример (некорректное использование).

```
#pragma omp paralllel for private(i) shared(a,xmax)
for(i=0; i<N; i++){
    if(a[i]>xmax)
#pragma omp critical
       xmax = a[i];
}// for
```

Пример (корректное использование, но не эффективное)

```
#pragma omp parallel for private(i) shared(a,xmax)
for(i=0; i<N; i++){
#pragma omp critical
    if(a[i]>xmax)
       xmax = a[i];
}// for
```

Атомарна секция

Барьеры

Барьеры – такой элемент синхронизации, который приостанавливает дальнейшее выполнение программы до тех пор, пока все потоки не достигнут этого барьера. Как только барьер достигнут всеми потоками, выполнение программы продолжается.

Синтаксис на C/C++

```
#pragma omp barrier
```

на Фортране

```
c$omp barrier
```

```
#pragma omp parallel
{
<инициализация>
#pargma omp barrier
<работа>
}
```

Фиксация порядка выполнения

Директива ordered в параллельных циклах (только там она может встречаться) говорит о том, что указанный блок должен исполняться в строго фиксированной последовательности. Внутри ordered секции одновременно может находиться только один поток.

Синтаксис на C/C++

```
#pragma omp ordered
    Структурный блок
```

на Фортране

```
c$omp ordered
    Структурный блок
c$omp end ordered
```

Пример.

```
#pragma omp parallel private(myid)
{
    myid = omp_get_thred_num();
#pragma omp for private(i)
    for(i=0; i<8; i++)
#pragma omp ordered
       printf("T%d: %d\n", myid, i);
}
```

Результат работы кода следующий:

```
T0: 0
T0: 1
T0: 2
T0: 3
T1: 4
T1: 5
T1: 6
T1: 7
```

Конструкция `flush`

Эта конструкция осуществляет немедленный сброс значений разделяемых переменных в память. Таким образом гарантируется, что во всех потоках значение переменной будет одинаковое. Неявно `flush` присутствует в следующих директивах: barrier, начале и конце критических секций, параллельных циклов, параллельных областей, single секций..

С ее помощью можно посылать сигналы потоком используя переменную как семафор. Когда поток видит, что значение разделяемой переменной изменилось, то это говорит, что произошло событие и следовательно можно продолжить выполнение программы далее. (Пример не работает. Не происходит блокирования)

Синтаксис:

```
#pragma omp flush(var1[, var2, ...])
c$omp flush(var1[, var2, ...])
```

Расширенные возможности OpenMP

В данном разделе пойдет речь об средствах OpenMP, которые позволяют более эффективно писать параллельные программы. К таким средствам относится директива threadprivate, которая позволяет один раз объявить приватную переменную для всех параллельных секций в рамках одного файла. Чтобы было возможно ее использовать, переменная должна быть объявлена как статическая, директива threadprivate должна присутствовать до объявления первой параллельной секции и количество потоков в программе должно быть постоянным.

Синтаксис:

```
#pragma omp threadprivate(var1[, var2 ...])
c$omp threadprivate(/cb1/[, /cb2/ ...])
```

Другим расширением OpenMP является возможность использовать синхронизацию потоков по средством блокировок. Блокировки в OpenMP аналогичны мутексам в POSIX threads. Даже набор функций для работы с ними аналогичен.

void omp_init_lock(omp_lock_t *lock)

инициализирует блокировку и связывает ее с параметром lock.

void omp_destroy_lock(omp_lock_t *lock)

деинициализирует переменную, связанную с параметром lock.

void omp_set_lock(omp_lock_t *lock)

Блокирует выполнение потока до тех пор пока блокировка на переменную lock не станет доступной.

void omp_unset_lock(omp_lock_t *lock)

Снимает блокировку с переменной lock.

void omp_test_lock(omp_lock_t *lock)

Пытается установить блокировку и если операция выполнена удачно, возвращает не нулевое значение. В противном случае возвращается ноль. Функция не блокирующая.

Прототипы функций описаны в omp.h

```
...
#include <omp.h>
void main(){
   omp_lock_t lock;
   int i, p_sum = 0, res = 0;

   omp_init_lock(&lock);
#pragma omp parallel firstprivate(p_sum)
  {
#pragma parallel for private(i)
     for(i=0; i<100000; i++)
         p_sum +=i;
     omp_set_lock(&lock);
     res += p_sum;
     omp_unset_lock(&lock);
  }
  omp_destroy_lock(&lock);
  printf("%d\n", res);
}
```

Отладка OpenMP кода

Настройка производительности OpenMP кода

Будем считать, что вопрос о том стоит ли оптимизировать программу или нет не стоит. Предположим, что вас не устраивает производительность и Вы решили заняться оптимизацией.

Оптимизацию любой программы стоит начинать уже на стадии выбора алгоритма поскольку именно за счет правильно выбранного алгоритма можно получить прирост производительности на поряки. Чуть менее значимый вклад дает оптимизация реализации и распараллеливание. В рамках текущего курса – это использование OpenMP. Поэтому прежде чем распараллеливать программу с OpenMP настоятельно рекомендуется добиться максимальной производительности последовательной версии.

Основной подход

Исход из общих соображений можно предложить следующий подход:

- Использовать автоматическое распараллеливание средствами компилятора.
- С помощью профилировщика выявить участки кода, которые наиболее требовательны к процессорному времени.
- Добавить директивы OpenMP для наиболее важных циклов.
- Если такое распаралеливание не дало ожидаемого прироста производительности, то выполнить проверку на
    - стоимости порождения процессов
    - размер циклов
    - балансировку загрузки
    - количество ссылок на разделяемые переменные
    - излишнюю синхронизацию
    - стоимость доступа к памяти

Рассмотрим некоторые моменты более подробно.

Автоматическое расспаралеливание

Многие компиляторы, которые поддерживают OpenMP позволяют производить автоматическое распараллеливание программ. При этом распараллелины могут быть только циклы, в которых компилятор не нашел зависимости по итерациям. Анализируя код компилятор сам вставляет в программу директивы OpenMP. В случае успеха пользователь уведомляется о том, что цикл был распараллелен, если нет, то сообщается почему.

```
$> icc -parallel -par_report3 text.c
test.c(61) : (col. 5) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 61
      shared: {"A", "B"}
      private: {"i", "j"}
      first private: { }
      reductions: { }
   procedure: chk_bk
   serial loop: line 74: not a parallel candidate due to insufficent work
   serial loop: line 82: not a parallel candidate due to insufficent work
   serial loop: line 66
      anti data dependence assumed from line 68 to line 68, due to "B"
```

Здесь для компилятора Intel ключ `-parallel` указывает на необходимость выполнит автоматическое расспаралеливание, ключ `-par_report` говорит о том, что необходимо

выводить отчетность по распараллеливанию. Число в конце ключа – уровень отчетности.

Иногда компилятор может предполагать наличие зависимости по данным в цикле, которых реально нет, и как результат, отказываться от его распараллеливания. В этих случаях компилятору можно помочь, указав, что в этом цикле итерации независимы.

Example

Профилирование программы

Иерархия памяти

Большинство вычислительных систем имеют следующую иерархию памяти:

1. регистры
2. кэш первого уровня
3. кэш второго уровня
4. локальная память
5. удаленна память (например, память другого узла кластера или жесткий диск)

При этом чем ниже по списку, тем большее время требуется на извлечение данных из соответствующей памяти.

Следовательно, в целях оптимизации надо более эффективно использовать локальную память, кэш и минимизировать обращения к удаленной памяти. Для этого надо стараться размещать данные в памяти так, чтобы обращение к ним происходило с минимальным количеством переписываний или вообще без переписывания кэша. Т.е. доступ к элементам массива должен осуществляться в той последовательности, в которой они лежат в памяти. Так, при работе с многомерными массивами в C/C++ наиболее быстро будет происходить доступ к элементам по самому правому (по записи) индексу, а в Фортране по самому левому. Иногда, для оптимизации работы с памятью следует поменять местами вложенные циклы.

```
for(j=1; j<M-1; j++)
  for(i=1; i<N-1; i++)
    B[i][j] = (A[i][j-1] + A[i][j+1] + A[i-1][j] + A[i-1][j])/4.0;
```

↓

```
for(i=1; i<N-1; i++)
  for(j=1; j<M-1; j++)
    B[i][j] = (A[i][j-1] + A[i][j+1] + A[i-1][j] + A[i-1][j])/4.0;
```

Задачи

Задача 1

Написать программу где каждый поток печатает свой идентификатор, количество потоков

всего и строчку «Hello World». Запустить программу с 8 потоками. Всегда ли вывод идентичен? Почему?

Задача 2

Написать программу, в которой объявлен массив из 16000 элементов и инициализирован так, что значение элемента массива равно его порядковому номеру. Затем создайте результирующий массив, в котором (за исключением крайних элементов) будут средние значения исходного массива:

b[i] = (a[i-1] + a[i] + a[i+1])/3.0

Запустите программу с 8-ю процессами при различных типах распределения работ.

Задача 3

Модифицируйте задачу 1 так, что бы потоки распечатывали свои идентификаторы в обратном порядке. Существует как минимум 5 способов решения. Постарайтесь найти как можно больше.

Задача 4

Напишите программу перемножения больших матриц. Сравните врем выполнения последовательной и параллельной программы на 2,4, 8 потоках (процессорах).

Задача 5

Напишите программу, которая читает из файла координаты точек в 3D пространстве (x,y,z) и вычисляет геометрический центр, который есть среднее по x, y и z. Напишите две версии программы: одну с расспараллеливанием цикла, другую с функциональной декомпозицией.

Задача 6

Используя функциональную декомпозицию перепишите задачу 5 для вычисления значения по формуле:

$$(\sum x + \sum y + \sum z)/3\mathrm{N}$$

где N – количество точек. Используйте глобальную сумму и критические секции.