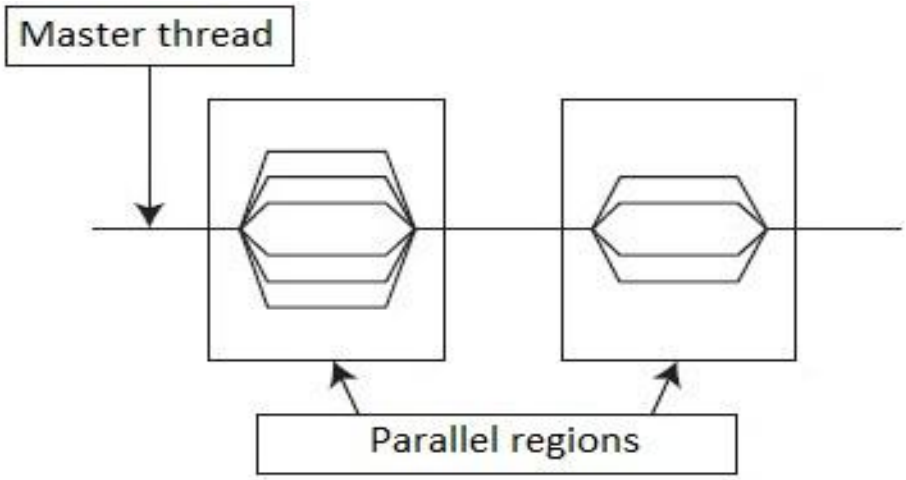
Введение в технологию OpenMP

- общая концепция.
- сравнение MPI и OpenMP
- организация параллельных и последовательных секций
- распределение работы между нитями
- синхронизация нитей
- работа с общими и локальными данными
- плюсы и минусы

Технология OpenMP

- Одно из наиболее популярных средств программирования компьютеров с общей памятью (в том числе многоядерных ПК).
- Базируется на традиционных языках программирования и использовании специальных псевдокомментариев.
- Реализован привлекательный с точки зрения пользователя подход к распараллеливанию: проблема возлагается на компилятор, а пользователь только выдает компилятору ценные указания: «вот это можно исполнить параллельно, а вот это – нельзя!» Ценные указания оформляются как псевдокомментарии в тексте программы.
- За основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляется ***набор директив, процедур и переменных окружения*****.** Стандарт OpenMP разработан для языков Фортран, C и C++. Все основные конструкции для этих языков похожи.
- **OpenMP работает в системах с общей памятью и основан на понятии «легковесного процесса» или нити (thread).**

OpenMP: последовательные и параллельные секции



- Весь текст программы разбит на последовательные и параллельные области.
- В начальный момент времени порождается нить-мастер (**master thread**), которая начинает выполнение программы со стартовой точки.
- Основная нить и только она исполняет все последовательные области программы.
- Для поддержки параллелизма используется схема Fork/Join (ветвление\объединение). При входе в параллельную область порождаются дополнительные нити (Fork). После рождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области.
- При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она (Join).

Сравнение OpenMP и MPI

- В отличие от MPI, реализующей модель распределенной памяти, OpenMP ориентирована на компьютерные системы с общей памятью, где доступ к любой ячейке памяти имеют все доступные узлы
- В отличие от MPI, где каждый параллельный процесс имеет до-ступ только к своей локальной памяти, в технологии OpenMP взаимодействие между параллельными потоками (нитями, threads) осуществляется за счет общих переменных (shared).
- В отличие от MPI, где все операции компьютерного кода выполняют все задействованные параллельные процессы, в OpenMP работу начинает одна нить (нить-мастер). Параллельные области организуются с помощью специальных OpenMP-директив.
- В отличие от MPI, реализующей явный параллелизм, в OpenMP имеются конструкции высокоуровневого параллелизма для распараллеливания циклов и независимых фрагментов.
- В отличие MPI, реализованной в виде библиотеки функций, синтаксис OpenMP главным образом основан на директивах (хотя функции также присутствуют)

OpenMP: общие и локальные переменные

- В параллельной области все переменные программы разделяются на два класса: общие (SHARED) и локальные (PRIVATE).
- **Общая переменная** всегда существует лишь в одном экземпляре для всей программы и доступна всем нитям под одним и тем же именем.
- Объявление **локальной переменной** вызывает порождение своего экземпляра данной переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.
- Можно указать, что по умолчанию (default) тип будет private
- Firstprivate: присутствуют в последовательной части кода, предшествующей параллельной секции. В параллельной секции это значение присваивается локальным переменным с тем же именем в каждой нити.
- Lastprivate: после завершения блока переменная сохраняет значение, полученное в завершившемся самым последним парараллельном потоке

Модель OpenMP

- Рассмотренные два понятия: ***области и классы переменных***, – определяют идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляется параллельными областями; именно эти области, и только они, исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными.
- Число нитей можно установить внутри программы либо поменять «снаружи» с помощью команды:

  export OMP_NUM_THREADS=256
- Возможны вложенные параллельные секции. В зависимости от реализации – число вложений тоже можно поменять (OMP_NESTED)
- Фортран: директивы OpenMP располагаются в комментариях и начинаются с одной из комбинаций: !OMP, COMP или *$OMP.
- C\C++: используются директивы прагмы **#pragma OMP**.

OpenMP: как организовать параллельные области

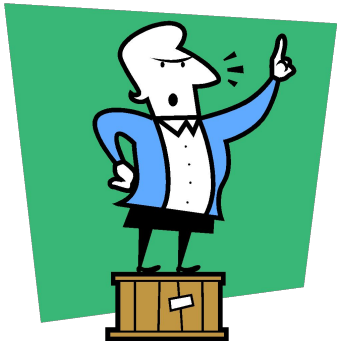
Выделение параллельного блока программы:

#pragma OMP parallel [описание переменных]

{ структурный блок программы}

- Для выполнения кода в фигурными скобках, порождается OMP_NUM_THREADS-1 нитей.
- Процесс, выполнивший директиву PARALLEL (нить-мастер), всегда получает номер 0.
- OMP_NUM_THREADS – переменная окружения, определяющая количество потоков (нитей) в параллельной программе.
- Все нити исполняют код параллельной секции, после чего автоматически происходит неявная синхронизация всех нитей.
- Как только все нити доходят до этой точки, нить-мастер продолжает выполнение последующей части программы, а остальные нити уничтожаются.

OpenMP: Пример



C\C++:

```
#pragma omp parallel [options, variables]
{ < parallel block > }

#include <omp.h>
main () {
// Serial code
…
// open parallel section
#pragma omp parallel
{
...
}
// close parallel section
// serial code
…
}
```

```
#include <stdio.h>
int main(){
printf("SERIAL 1\n");
#pragma omp parallel
{
printf("PARALLEL \n");}
printf(" SERIAL 2\n");
return 0;}
```

- Нить-мастер печатает «**SERIAL** 1»
- По директиве **parallel** порождаются новые нити, каждая из которых напечатает текст «**PARALLEL**».
- Затем параллельная секция завершается, и оставшаяся нить-мастер напечатает «**SERIAL** 2».

OpenMP:
явный параллелизм

- Функция omp_get_thread_num возвращает номер нити.
- Функция omp_get_num_threads возвращает кол-во нитей.
- Функция omp_set_num_threads устанавливает кол-во нитей.

Необходимость порождения нитей может определяться динамически с помощью опции IF в директиве OMP PARALLEL IF(<условие>)

Пример явного параллелизма:

Ветвление в зависимости от номера параллельного потока.

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

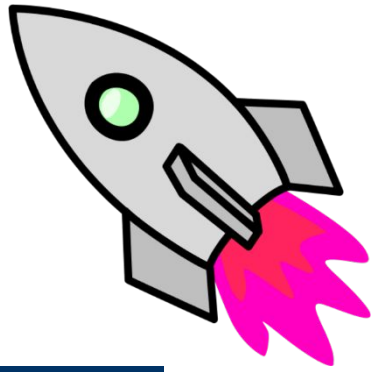
Example

```
#include <omp.h>
#include <stdio.h>
int main()
{
int nthreads, tid;
// open parallel section
omp_set_num_threads(3)
#pragma omp parallel private(nthreads, tid)
{
// each thread determines its number
tid = omp_get_thread_num();
printf("Hello World from the thread = %d\n", tid);
// 0-thead prints  quantity of theads
      if (tid == 0)
      {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
      }
}
// end parallel section
printf("Only Master-thread continues execution \n");
return 0;
}
```

OUTCOME:

Hello World from the thread = 0
Number of theads=3
Hello World from the thread = 1
Hello World from the thread = 2

Запуск OpenMP-программы на HybriLIT



Загрузкамодуля (GNU или Intel):

GNU: **default (v.4.4.7)**

module add hlit/gcc/4.9.3 (или 4.8.4 или др.)

Intel: **module add intel-2016.1.150** (или др.)

Компиляция:

GNU: gfortran -fopenmp example.f gcc -fopenmp example.c

Intel: ifort -qopenmp example.f icc -qopenmp example.c

Интерактивный запуск
(< 5 min)
./a.out

Batch-запуск:
sbatch script_omp

File script_omp:

```
#!/bin/sh
#SBATCH -p cpu
#SBATCH -c 5
#SBATCH –t 10
export OMP_NUM_THREADS=5
export OMP_PLACES=cores
./a.out
```

Как установить число нитей

1. Функция omp_set_num_theads (установка числа нитей во всех параллельных секциях кода)

```
omp_set_num_threads(2);
#pragma omp parallel
{ … }
```

2. Опция pragma omp parallel (установка числа нитей в данной параллельной секции)

```
#pragma omp parallel num_threads(3)
{ … }
```

3. Системная команда

```
export OMP_NUM_THREADS=5
```

Пример: функция omp_set_num_threads(); опция num_threads

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
omp_set_num_threads(2);
#pragma omp parallel
     num_threads(3)
{
printf("Параллельная область
     1\n");
}
#pragma omp parallel
{
printf("Параллельная область
     2\n");
return 0;}
```

- Перед 1ой параллельной областью вызов **omp_set_num_threads(2)** выставляет кол-во нитей, равное **2**.
- **НО!** к ней применяется опция **num_threads(3)**, поэтому сообщение **"**Параллельная область 1**"** напечатают **три** нити.
- К 2ой параллельной области опция **num_threads** не применяется, поэтому действует значение, установленное **omp_set_num_threads(2).**
- Поэтому сообщение **"**Параллельная область **2"** будет выведено двумя нитями.

Пример: применение опции reduction

Подсчет общего количества порождённых нитей.

```
#include <stdio.h>
int main()
{
int count = 0;
#pragma omp parallel reduction (+: count)
{
count++;
printf("Текущее значение
count: %d\n",count);
}
printf("Число нитей: %d\n", count);
return 0;}
```

- Каждая нить инициализирует локальную копию переменной **count = 0**.
- Далее, каждая нить увеличивает значение своей копии переменной **count** на 1 и выводит полученное число.
- На выходе из параллельной области происходит суммирование значений переменных **count** по всем нитям.
- Полученная величина становится новым значением переменной **count** в последовательной области.

Высокоуровневый параллелизм: pragma omp sections

```
#pragma omp parallel
{
#pragma omp sections
    {
#pragma omp section
    { block 1 }
#pragma omp section
    { block 2 }
    ..............
    }
}
```

Если два или более фрагмента информационно независимы, их можно исполнять в любом порядке, в частности, параллельно друг другу. При использовании прагмы omp sections каждый из таких фрагментов будет выполнен какой-либо одной нитью.

Высокоуровневый параллелизм: параллельное выполнение цикла

Если в параллельной области встретился оператор цикла, каждая нить выполнит все итерации данного цикла.

Для распределения итераций цикла между различными нитями нужно использовать директивы !$OMP DO (фортран) и **#pragma omp for (C\C++),** которые относятся к идущему следом оператору цикла.

Пример:

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

По умолчанию в конце цикла реализуется барьерная синхронизация, которую можно отменить с помощью опции nowait

Open MP: синхронизация нитей (1)

Директива SINGLE используется для однократного выполнения части кода, **е**сли в параллельной секции какой-то участок кода должен быть выполнен лишь один раз.

#pragma omp single{ }

Такой участок кода будет выполнен (единожды) нитью, первой дошедшей до данной точки программы.

БАРЬЕР оформляется с помощью директивы

#pragma omp barrier

Все нити, дойдя до этой директивы, останавливаются и ждут пока остальные нити не дойдут до этой точки программы, после чего все нити продолжают работать дальше.

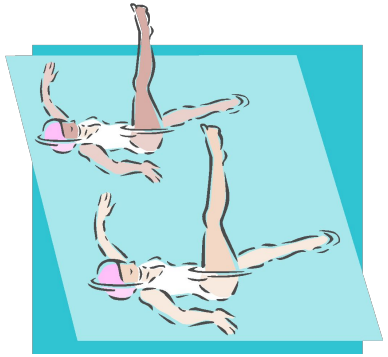
Директива MASTER выделяет участок кода, который должен быть выполнен только нитью-мастером.

#pragma omp master

{ структурный блок программы}

Остальные нити пропускают данный участок.

Open MP: синхронизация нитей (2)



Критическая секция оформляется с помощью директивы

#pragma omp critical [name]

{ структурный блок программы}

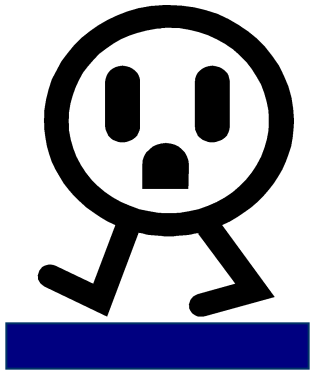
В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока указанная нить не закончит выполнение данной критической секции. Как только нить закончит выполнение критической секции, одна из заблокированных на входе нитей войдет в секцию. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Директива atomic относится к идущему непосредственно за ней оператору. Частый случай использования критических секций. Пример:

#pragma omp atomic

Expr++;.

Example:
pragma omp critical



```
#include <stdio.h>
#include <omp.h>
int main()
{
  int sum = 0;
  int expr = 0;
  #pragma omp parallel num_threads(3) private(expr) shared(sum)
  {
  expr =omp_get_thread_num()+1;
  printf("in thread number %d expr = %d \n", omp_get_thread_num(),expr);
  #pragma omp critical
  {
  sum += expr;
  }
  printf("after critical section\n");
  printf("In thead number %d: expr = %d, sum = %d\n",
      omp_get_thread_num(), expr, sum);
  }
  printf("After the parallel section closing: sum = %d\n", sum);
return 0;
}
```

Open MP: синхронизация нитей (3)

Согласование памяти. Синхронизация этого типа используется для обновления локальных переменных в оперативной памяти.

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени каждая нить будет видеть единый согласованный образ памяти. Именно для этих целей и предназначена данная директива;

#pragma omp flush (var1, [var2,…])

Ее выполнение предполагает, что значения всех переменных, временно хранящиеся в регистрах, будут занесены в основную память, все изменения переменных, сделанные нитями во время их работы, станут видимыми остальным нитям, если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. Таким образом, после выполнения данной директивы все перечисленные в ней переменные имеют одно и то же значение для всех параллельных потоков. Выполнение данной директивы в полном объеме может повлечь значительные накладные расходы.

OpenMP: плюсы и минусы

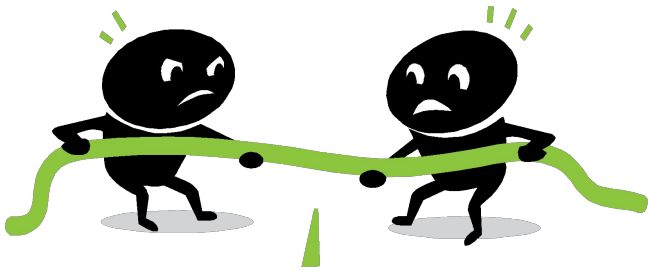
Технология изначально спроектирована, чтобы пользователь мог работать с единым текстом для параллельной и последовательной программ. Обычный компилятор на последовательной машине директивы OpenMP просто «не замечает».

Другое достоинство OpenMP – возможность постепенного распараллеливания программы. Взяв за основу последовательный код, пользователь step-by-step добавляет новые директивы, описывающие новые параллельные секции. Тем самым, упрощается процесс создания кода.

С другой стороны, для ускорения вычислений необходимо, чтобы трудоемкость параллельных процессов существенно превосходила бы трудоемкость порождения потоков. Операции синхронизации и инициализации параллельных потоков эквивалентны примерно трудоемкости выполнения 1000 арифметических операций. Поэтому при выделении параллельных областей и разработке параллельных процессов необходимо, чтобы их трудоемкость была не менее 2000 операций.

Важно следить за согласованием памяти! Возможность data race

OpenMP: Пример data racing



```
#pragma omp sections
{
# pragma omp section
{…
C=A+B;
…}
#pragma omp section
{…
B=C+A;
…}
#pragma omp section
{…
A=B+C;
…}
}
```

Эффект состязательности (race condition)

Результат зависит от того, в какой последовательности выполняются параллельные потоки.

Поскольку синхронизации нет, каждый раз будет разный результат.

При этом компилятор не выдает никакой диагностики.

В этом примере предполагается, что A,B,C – глобальные переменные (shared).