Open Multi-Processing

OpenMP - это стандарт, определяющий набор директив компилятора, библиотечных процедур и переменных среды окружения для создания многопоточных программ использующих общую память.

Стандарт предусматривает использование OpenMP для алгоритмических языков C90, C99, C11, C++.

Цели OpenMP

Стандартизация:
- Единый стандарт для различных систем/архитектур/платформ с общей памятью.
- Определен совместными усилиями ведущих поставщиков программного и аппаратного обеспечения.

Краткость и выразительность:
- Небольшой набор директив и дополнительных элементов (clause).

Простота использования:
- Упрощенный механизм создания параллельных программ, практически не влияющий на работу программиста.

Переносимость:
- Поддерживается в C/C++ и Фортран.
- Поддерживается но множестве платформ (Unix/Linux, MacOS, Windows).

История OpenMP

- Начало разработки стандарта OpenMP, предназначалась для языка Fortran - 1997 год
- Intel, AMD*, ARM*, Cray*, IBM*, HP*, Micron*, NEC*, Nvidia*, Oracle* etc.
- Версии 1.0-2.5 (Октябрь 1997 – Май 2005)
- Первые версии, внедрение и развитие потокового распараллеливания циклов
- Версии 3.0, 3.1 (Май 2008 – Июль 2011)
- Добавление и развитие поддержки независимых задач
- Версия 4.0 (Июль 2013)
- Поддержка векторизации циклов (SIMD), поддержка ускорителей (target), зависимые задачи, встроенные механизмы обработки ошибок (cancel), пользовательские редукции, расширение атомарных конструкций
- Версия 4.5 (Ноябрь 2015), компиляторы GCC 6.0, Clang 3.8.
- Версия 5.1 (Ноябрь 2020), компиляторы GCC 11.0, Clang 12.0.
- Добавлена полная поддержка распараллеливания программ, написанных с использованием стандартов C11, C18, C++11, C++14, C++17 и C++20. Расширена поддержка специализированных аппаратных ускорителей.

Open Multi-Processing

Узнать версию стандарта возможно с помощью макроса _OPENMP. Данный макрос содержит 6 цифр в формате yyyy и mm, где первые 4 цифры это год, а две последующие это месяц создания формата.

```
#include <stdio.h>
int main(int argc, char** argv){
   #ifdef _OPENMP
      printf("OpenMP version is: %u.\n", _OPENMP);
   #else
      printf("OpenMP not exists.\n");
   #endif

   return 0;
}
```

Open Multi-Processing

Установка OpenMP для linux:

```
sudo apt-get install libomp-dev
```

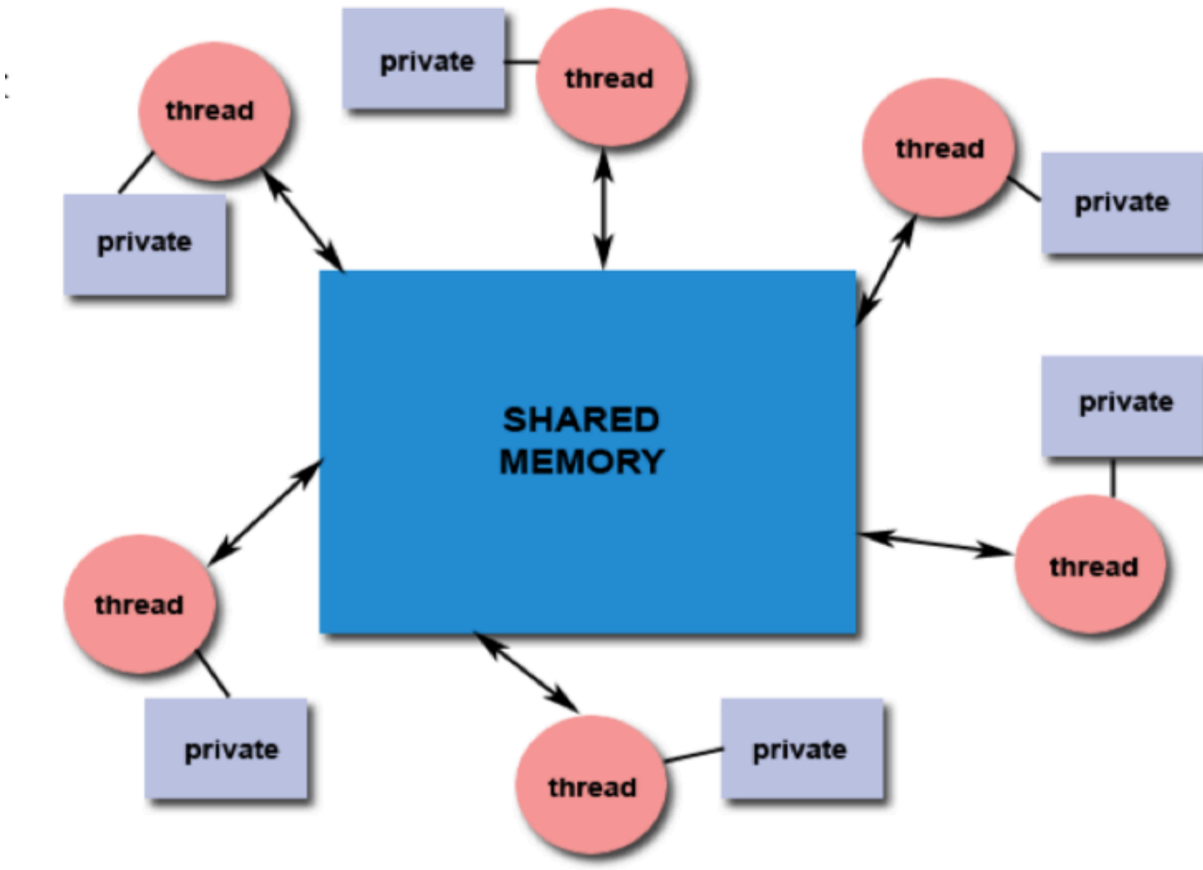
Чтобы собрать программу использующую `OpenMP` следует указать компилятору флаг `-fopenmp`, например:

```
gcc -fopenmp -o example example.c
```

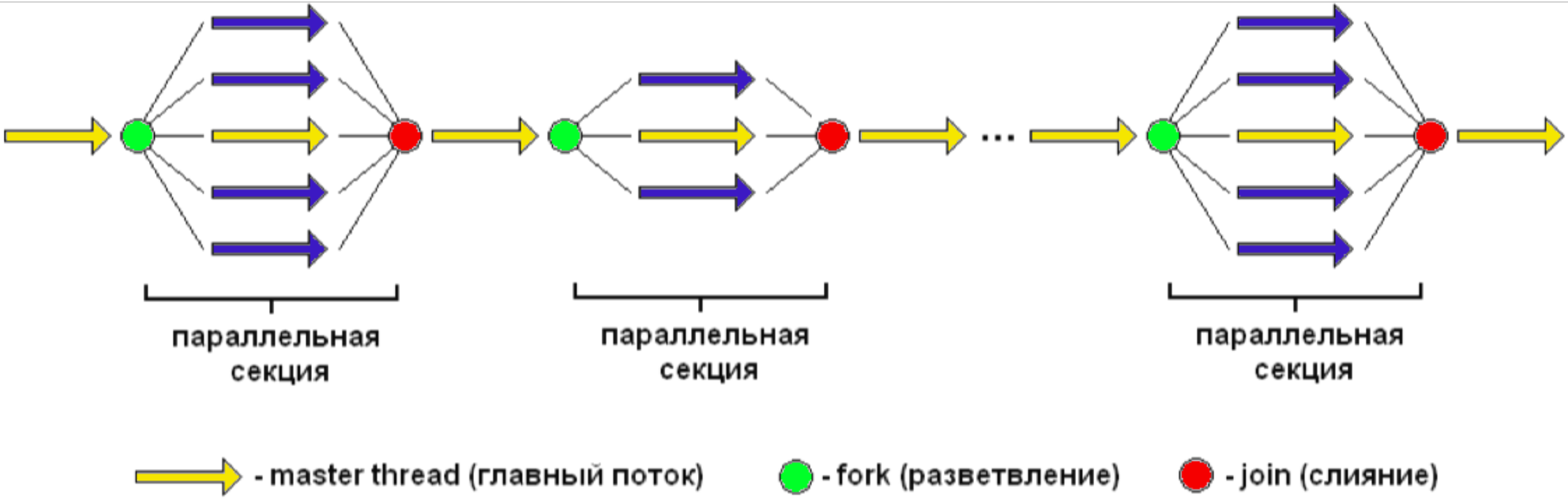
OpenMP использует подход SPMD (Single Program Multiply Data), то есть используется один и тот же код для описания поведения нескольких параллельных нитей.

Модель с разделяемой памятью

- Все потоки имеют доступ к глобальной общей памяти.
- Данные могут быть общие и приватные.
- Общие данные доступны всем потокам.
- Приватные данные доступны только одному потоку-владельцу.
- Требуется синхронизация для доступа к общим данным.



Open Multi-Processing



Программу написанную с помощью OpenMP чаще всего можно представить в виде набора последовательных блоков разветвлении и слияний.

Структура OpenMP-программы

Программа представляется в виде последовательных участков кода (`serial code`) и параллельных регионов (`parallel region`).

Каждый поток имеет номер `Thread Id`.
Мастер поток (`master`) имеет номер `0`.
Память процесса (`heap`) является общей для всех потоков.

`OpenMP` реализует динамическое управление потоками (`task parallelism`).

Модель параллельной программы

Последовательная область – один процесс (нить), вход в параллельную область – порождение некоторого числа процессов, их завершение, нить-мастер. Параллельные области могут быть вложенными друг в друга.

В отличие от полноценных процессов, порождение нитей является относительно быстрой операцией, поэтому частые порождения и завершения нитей не так сильно влияют на время выполнения программы.

Для написания эффективной параллельной программы необходимо, чтобы все нити были равномерно загружены, что достигается тщательной балансировкой загрузки при помощи механизмов OpenMP.

Необходимость синхронизации доступа к общим данным – существенна. Само наличие данных, общих для нескольких нитей, приводит к конфликтам при одновременном несогласованном доступе.

Значительная часть функциональности OpenMP предназначена для осуществления различного рода синхронизаций работающих нитей.

OpenMP не выполняет синхронизацию доступа различных нитей к одним и тем же данным. Пользователь должен явно использовать директивы синхронизации или соответствующие библиотечные функции. При доступе каждой нити к своей области памяти никакая синхронизация не требуется.

Когда использовать OpenMP?

Целевая платформа является многопроцессорной или многоядерной.

Если приложение полностью использует ресурсы одного ядра или процессора, то, сделав его многопоточным при помощи OpenMP, вы почти наверняка повысите его быстродействие.

Приложение должно быть кроссплатформенным.

OpenMP – кроссплатформенный и широко поддерживаемый API. А так как он реализован на основе директив pragma, приложение можно скомпилировать даже при помощи компилятора, не поддерживающего стандарт OpenMP.

Выполнение циклов нужно распараллелить.

Весь свой потенциал OpenMP демонстрирует при организации параллельного выполнения циклов. Если в приложении есть длительные циклы без зависимостей, OpenMP – идеальное решение.

Перед выпуском приложения нужно повысить его быстродействие.

Так как технология OpenMP не требует переработки архитектуры приложения, она прекрасно подходит для внесения в код небольших изменений, позволяющих повысить его быстродействие.

Однако OpenMP – не панацея от всех бед.

Технология ориентирована в первую очередь на разработчиков высокопроизводительных вычислительных систем и наиболее эффективна, если код включает много циклов и работает с разделяемыми массивами данных.

Open Multi-Processing

Для OpenMP переменные в параллельных секциях можно разделить на два основных класса:

`shared` (общие) - все нити видят одну и ту же переменную, которая хранится в общей памяти.

`private` (локальные) - для каждой нити существует экземпляр данной переменной.

Open Multi-Processing

По умолчанию все переменные, порожденные вне параллельной области при входе в неё являются общими. Исключения составляют счетчики итераций в цикле.

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

#define NUM_THREADS 5

int main(int argc, char** argv){
    int i = 10;
    int thread;
    omp_set_num_threads(NUM_THREADS);
    printf("значение счетчика i до выполнения параллельной секции : %d\n", i);
    #pragma omp parallel private(thread)
    {
        thread = omp_get_thread_num();
        #pragma omp for
        for(i = 0 ; i < NUM_THREADS; i++){
            printf("Номер нити: %d, значение счетчика i : %d\n", thread, i);
        }
    }
    printf("i после выполнения параллельной секции: %d\n", i);
    return 0;
}
```

Open Multi-Processing

Для OpenMP существует способ задать количество ожидаемых в параллельной части программы нитей. Это возможно сделать с помощью переменой среды OMP_NUM_THREADS:

```
export OMP_NUM_THREADS=n
```

или функции omp_set_num_threads() уже непосредственно из кода:

```
omp_set_num_threads(n);
```

Open Multi-Processing

Функции для работы с системным таймером:

```
double omp_get_wtime(void);
```

Возвращает астрономическое время в секундах, прошедшее с некого момента в прошлом.

```
double omp_get_wtick(void);
```

Возвращает разрешение таймера с сеундах (шаг).

Open Multi-Processing

Формат директив OpenMP может быть представлен в следующем виде:

```
#pragma omp <имя директивы> <параметры>
```

Начальная часть директивы `#pragma omp` не меняется, тип директивы определяется её именем. Каждая директива может сопровождаться произвольным количеством параметров. Для объявления директивы может быть использовано несколько строк программы, в этом случае используется знак обратного слеша. Директива, как правило, воздействует на следующий в программе оператор, который тоже может оказаться структурированным блоком кода.

Open Multi-Processing

Некоторые опции параллельной секции кода:

`if (условие)` - условие выполнения параллельной области.

`num_threads(целочисленное значение)` - позволяет задать напрямую количество нитей для параллельной области.

`default (shared | none)` - по умолчанию класс переменных для которых явно класс не задан.

`private (список)` - локальные переменные.

Open Multi-Processing

`firstprivate(список)` - переменные для которых порождается локальная копия в каждой нити. Локальные копии инициализируются значениями этих переменных в нити-мастере.

`shared(список)` - переменные, общие для всех нитей.

Open Multi-Processing

`reduction(оператор:список)` - позволяет собрать вместе в главном потоке результаты вычислений из параллельных потоков.
Оператор может принимать значения +, -, *, /, &, |, ^, &&. Действие выполняется после параллельной области. Так как стандарт не предусматривает фиксированную последовательность выполняемых действий, результат может варьироваться от запуска к запуску.

Open Multi-Processing

Пример, демонстрирующий выполнение двух последовательных и одну параллельную область.

```
#include <stdio.h>
int main(int argc, char** argv){
   printf(«Последовательная область 1\n»);
   #pragma omp parallel
   {
      printf(«Параллельная область\n»);
   }
   printf(«Последовательная область 2\n»);
   return 0;
}
```

Open Multi-Processing

Пример, демонстрирующий работу опции `reduction`.

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

Open Multi-Processing

Для выделения параллельных фрагментов программы следует использовать директиву `parallel`:

```
#pragma omp parallel [<параметр> …]
      <блок программы>
```

- для блока должно выполняться правило «один вход – один выход», не допускается передача управления извне в блок и из блока за пределы блока.

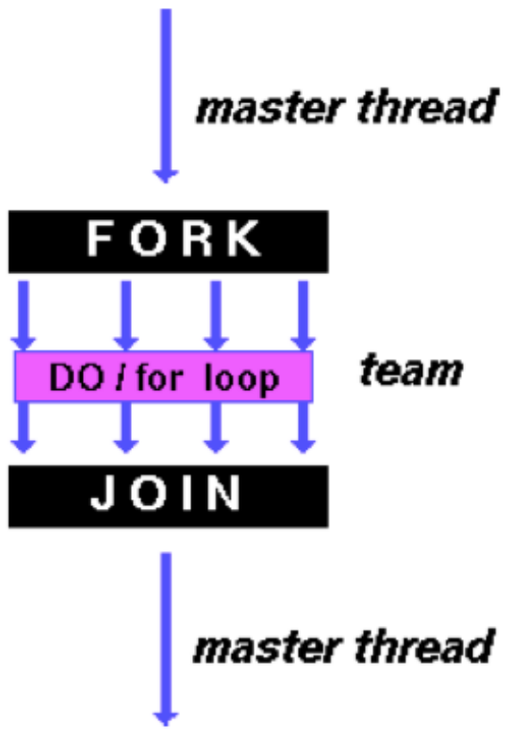
- по достижении директивы `parallel`, создается набор из `N` потоков, исходный поток программы является основным потоком этого набора (`master thread`) и имеет номер 0.

- код блока, следующий за директивой, дублируется или может быть разделен при помощи директив между потоками для параллельного выполнения.

- в конце блока директивы обеспечивается синхронизация потоков, выполняется ожидание (`join`) окончания вычислений всех потоков. Все потоки завершаются, дальнейшие вычисления продолжает выполнять только основной поток. В зависимости от среды потоки могут не завершиться, а ожидать дальнейших заданий.

Open Multi-Processing

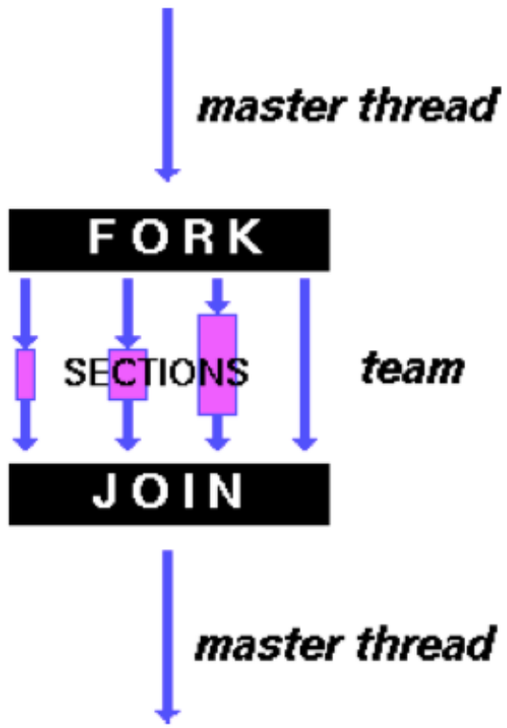
Так же существует несколько способов разделения работы между потоками.



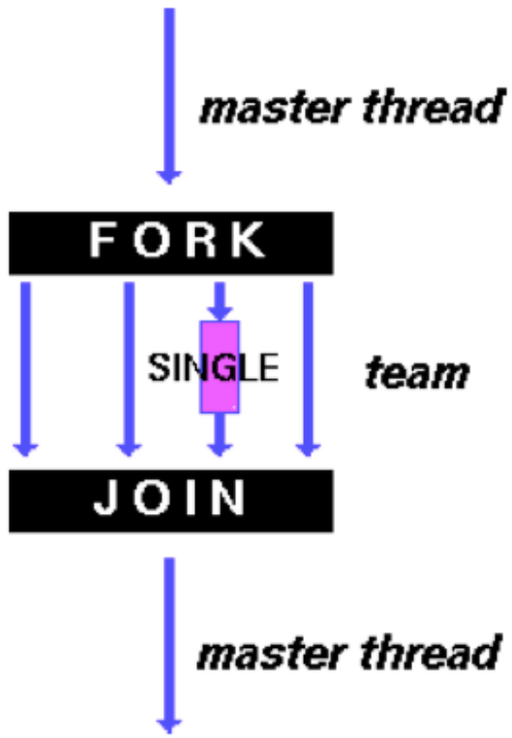

```
#pragma omp for
for (i=0;i<N;i++) {

// code

}
```




```
#pragma omp sections {
#pragma omp section
   // code 1
#pragma omp section
   // code 2
}
```




```
#pragma omp single {
// code
}
```