Технологии параллельного программирования на C++

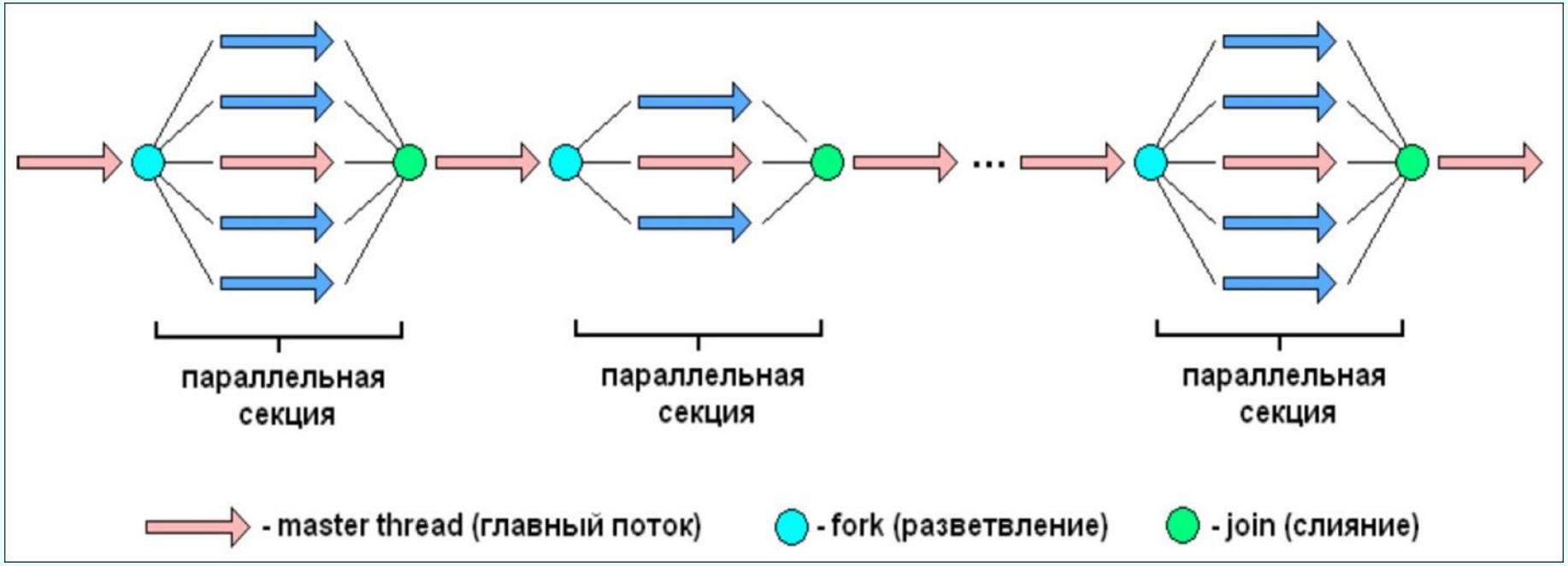
Семинар 5

OpenMP (Open Multi-Processing)

- 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
- можно рассматривать как высокоуровневую надстройку над Pthreads

Модель программирования

- Явное указание параллельных секций
- Fork – join параллелизм
- Поддержка динамических потоков
- Поддержка вложенного параллелизма



Компиляция программы

- **g++ test.cpp –fopenmp**
- Проверка того, что компилятор поддерживает какую-либо версию **OpenMP**

```
#include <iostream>
using namespace std;
int main(){
#ifdef _OPENMP
   cout<<"OpenMP is supported!";
#endif
}
```

- Задавать количество нитей **n** с помощью команды

  export OMP_NUM_THREADS=n

Директивы и функции

- **#pragma omp directive-name [опция[[,] опция]...]**
- Директивы
  - Определение параллельной области
  - Распределение работы
  - Синхронизация
- Чтобы задействовать функции библиотеки **OpenMP**, нужно задействовать заголовочный файл **<omp.h>**

Замер времени

- **double omp_get_wtime(void);**

  Функция возвращает в вызвавшей нити время в секундах

- **double omp_get_wtick(void);**

  Функция возвращает в вызвавшей нити разрешение таймера в секундах

Директива parallel

- **#pragma omp parallel [опция[[,] опция]...]**
- **Опции**
  - **if(условие)**

    выполнение параллельной области по условию
  - **num_threads(целое число)**

    явное задание количества нитей
  - **private(список)**

    задает список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных не определено
  - **firstprivate(список)**

    задает список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере
  - **shared(список)**

    Задает список переменных общих для всех нитей
  - **reduction(оператор:список)**

    Задает оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; над локальными копиями выполняется заданный оператор (+, -, *, &,|,…)
  - **Copyin(список), lastprivate(список), default(shared|private|none)**

Параллельная область

- При входе в параллельную область порождаются новые **OMP_NUM_THREADS-1** нитей, каждая нить получает свой уникальный номер

- Порождающая нить получает номер **0 и становится основной нитью группы («мастером»).**
- Остальные нити получают в качестве номера целые числа с **1 до OMP_NUM_THREADS-1.**

```
#include <iostream>
using namespace std;
int main()
{
   cout<<"Sequential section 1"<<endl;
   #pragma omp parallel
   {
      cout<<"Parallel section"<<endl;
   }
   cout<<"Sequential section 2"<<endl;
}
```

Опция reduction

```
#include <iostream>
using namespace std;
int main()
{
    int count = 0;
    #pragma omp parallel reduction (+: count)
    {
        count++;
        cout<<count<<endl;
    }
    cout<<"Number of threads:"<<count<<endl;
}
```

Вспомогательные функции

- **void omp_set_num_threads(int num);**

  Задает количество нитей
- **int omp_get_thread_num(void);**

  Определяет номер нити

  в текущей параллельной секции

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
   omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
   {
      cout<<"1 ";
   }
#pragma omp parallel
   {
      cout<<"2 ";
   }
}
```

Вспомогательные функции

- **void omp_set_dynamic(int num);**

  num (0 или 1). Если num =1 (значение переменной **OMP_DYNAMIC**),

  то система может динамически изменять количество нитей

- **int omp_get_dynamic(void);**

  Узнать значение переменной **OMP_DYNAMIC**

- **int omp_get_max_threads(void);**

  Возвращает максимально допустимое число нитей для использования в следующей параллельной области

- **int omp_get_num_procs(void);**

  Возвращает количество процессоров, доступных для использования программе пользователя на момент вызова

Директива master

- **#pragma omp master**

  Выделяет участок кода, который будет выполнен только нитью-мастером

Директива single

- **#pragma omp single [опция [[,] опция]...]**

  Позволяет в параллельной часть кода выполнять выделенный участок лишь один раз

- Опции
  - **private(список), firstprivate(список), copyprivate(список)**
  - **nowait**

  После выполнения выделенного участка происходит неявная барьерная синхронизация параллельно работающих нитей. Опция nowait позволяет нитям, уже дошедшим до конца участка, продолжить выполнение без синхронизации с остальными

Пример программы с опцией nowait

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
#pragma omp parallel num_threads(2)
    {
        cout<<" message_1 ";
#pragma omp single nowait
        {
            cout<<omp_get_thread_num();
        }
        cout<<" message_2 ";
    }
}
```

Параллельные секции

- **#pragma omp sections [**опция **[[,]** опция**]...]**
- Эта директива определяет набор независимых секций кода, каждая из которых выполняется своей нитью.
- Директива **section** задаёт участок кода внутри секции **sections** для выполнения одной нитью:
- **#pragma omp section**

Параллельные циклы

- **#pragma omp for [опция [[,] опция]...]**
- Распределяет итерации цикла между различными нитями
- Опции
  - **private(список), firstprivate(список)**, **lastprivate(список), reduction(оператор:список), collapse(n), ordered, nowait**
  - **schedule(type [, chunk])** – задает, каким образом итерации цикла распределяются между нитями

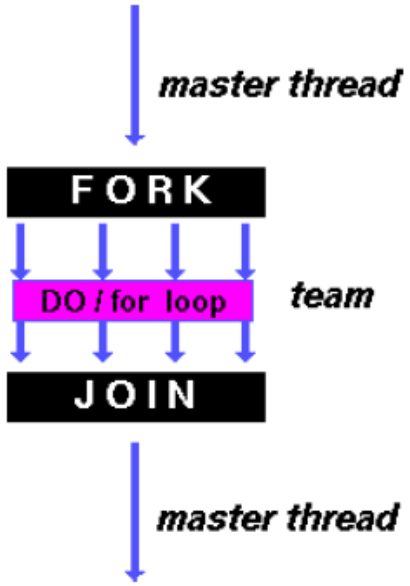
Опция schedule(type [, chunk])

- Параметр **type**
  - **static** – блочно-цикличное распределение итераций цикла, размер блока – chunk.
  - **dynamic** – динамическое распределение итераций с фиксированным размером блока. Каждая нить получает chunk итераций. Нить, которая заканчивает выполнение своей порции, получает новую порцию из chunk итераций
  - **Auto** – способ распределения итераций выбирается компилятором и/или системой выполнения

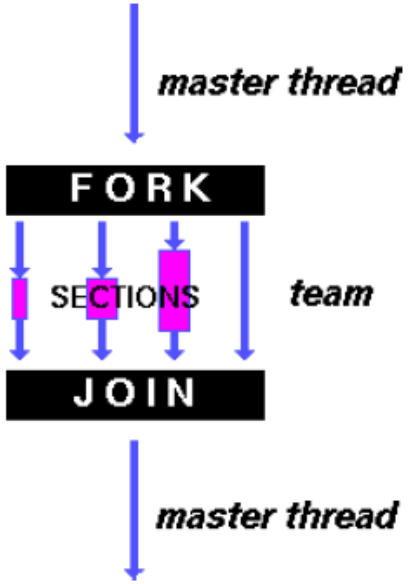
Пример с опцией for

```
#include <iostream>
#include <unistd.h>
#include <omp.h>
using namespace std;
int main()
{
   int i;
#pragma omp parallel private(i)
   {
#pragma omp for
      for (i=0; i<36; i++)
      {
         cout<<omp_get_thread_num();
         sleep(1);
      }
   }
}
```

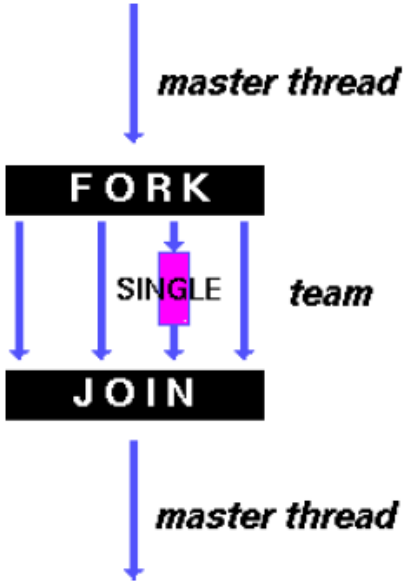
```
#pragma omp for
  for (i=0;i<N;i++)
  {
    // code
  }
```




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




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



Синхронизация

- **#pragma omp barrier**

  Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше

- **#pragma omp critical [(<имя_критической_секции>)]**

  В каждый момент времени в критической секции может находиться не более одной нити.

  Все критические секции, имеющие одно и тоже имя, рассматриваются единой секцией

Порядок создания параллельных программ

1. Написать и отладить последовательную программу
2. Дополнить программу директивами OpenMP
3. Скомпилировать программу компилятором с поддержкой OpenMP
4. Запустить программу

Задача 1

- Написать программу, которая складывает два вектора (одномерные массивы большой размерности)

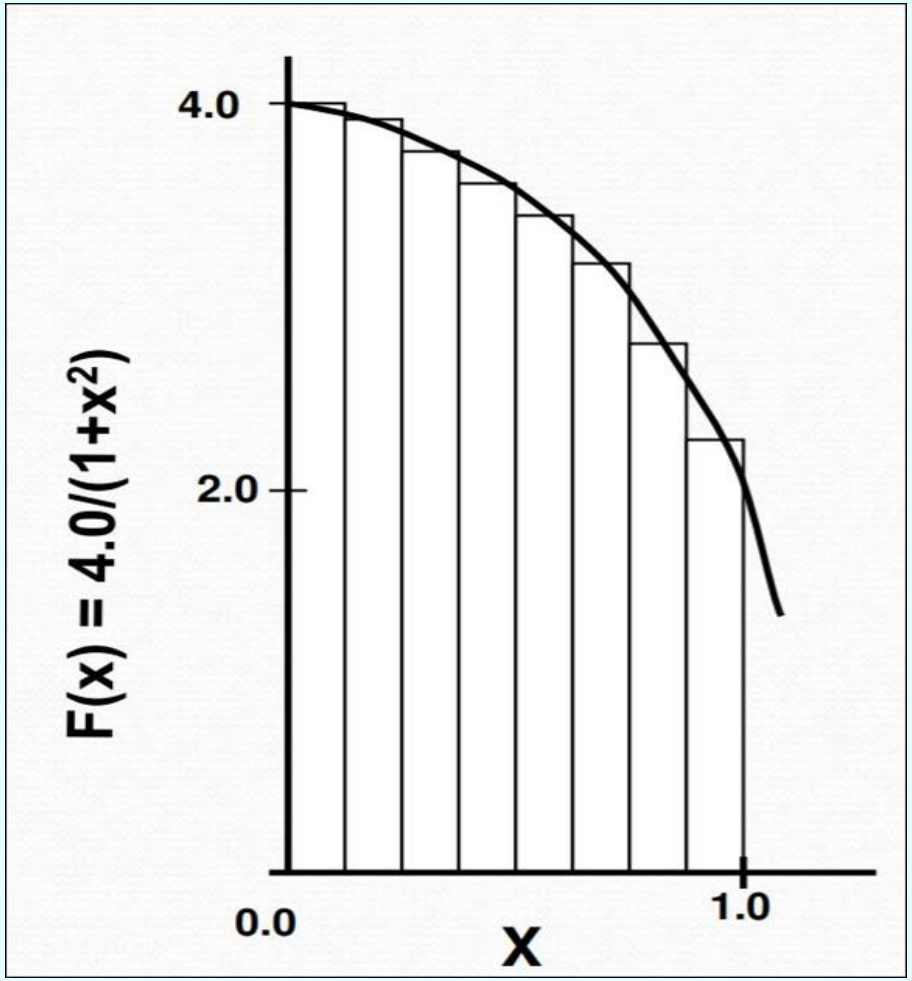
Задача 2

- Написать программу, которая делает скалярное произведение 2 векторов (одномерные массивы большой размерности)

- Построить графики ускорения и эффективности

Задача 3

- Написать программу, которая вычисляет интеграл (с мелким разбиением отрезка)



$$\int_0^1 \frac{4.0}{(1+x^2)} \, dx = \pi$$

- Построить графики ускорения и эффективности

Задача 4

- Написать программу, в которой задается массив. Нужно найти количество элементов кратных 6.