Open MP

Хотим все, сразу и бесплатно

- Программисты всегда мечтали, что в их программы параллелизм проберется сам, незаметно и не отвлекая их от более важных дел.

OpenMP

- Открытый стандарт OpenMP Architecture Review Board
- Windows, Linux, Mac OS X, Solaris
- «Учредители»:

Поддержка OpenMP

- Microsoft Visual Studio 2005 и 2008 поддерживает OpenMP 2.0 в редакциях Professional и Team System, 2010 - в редакциях Professional, Premium и Ultimate, 2012 - во всех редакциях.
- GCC 4.2 поддерживает OpenMP
- Intel C++ Compiler поддерживает версию OpenMP 3.0.

Что такое OpenMP?

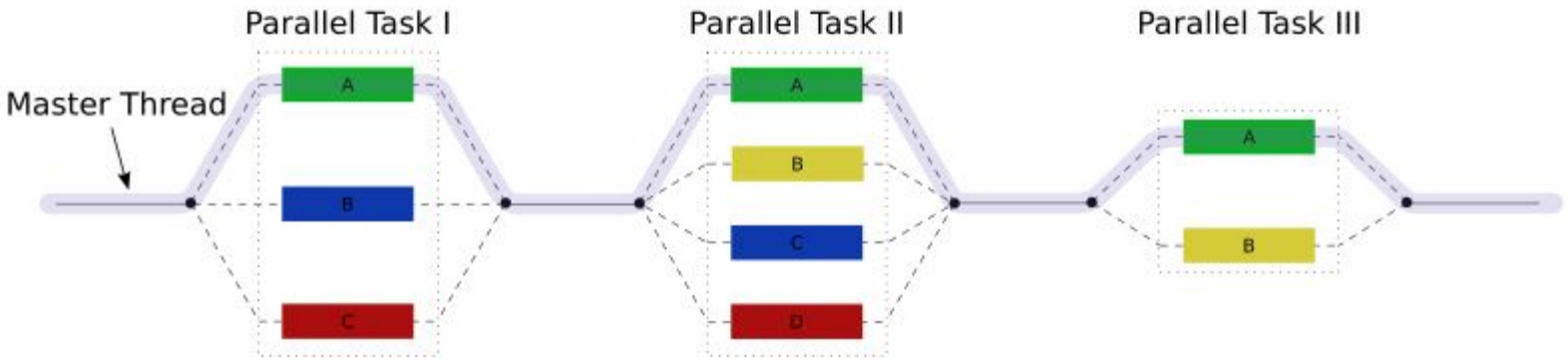
- Стандарт
- Библиотека времени выполнения
- Поддержка в компиляторе

Что такое OpenMP?

- OpenMP подразумевает маркировку параллельного кода специальными директивами. Параллелизация происходит «почти автоматически».
- Паралелльные участки кода соседствуют с последовательными
- 1997 г – Fortran,
- 1998 г – C++
- ЧУДО!!!

OpenMP-программа

- Типичная схема



- Главный поток создает несколько вспомогательных и заставляет их считать задачу параллельно

Лучше один раз увидеть…

```
double a[N], b[N], c[N]; int i;
omp_set_dynamic(0);
omp_set_num_threads(10);
#pragma omp parallel shared(a, b, c) private(i)
{
#pragma omp for
    for (i = 0; i < N; i++)
      c[i] = a[i] + b[i];
}
```

Формат директив

- #pragma omp [directive] [clause [clause] …]
  - [directive] – название директивы
  - [clause] - условие
- Продление – «\» в конце

- #pragma omp parallel private (i, j) \
  shared (a, b, c)

Подключение в C++

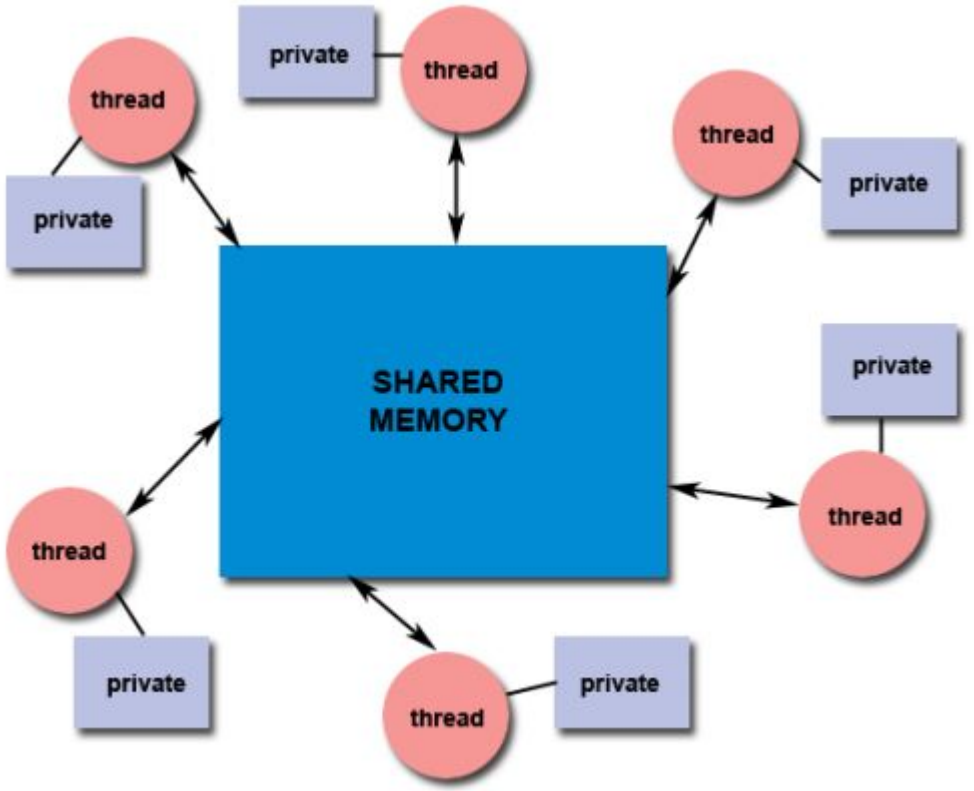
- /openmp – флаг компилятору в VS

Инкрементальный параллелизм

- Не нужно выкидывать старую программу и писать новую
- Можно постепенно переписывать код на использование параллельных вычислений

Модель с разделяемой памятью

- Все потоки имеют доступ к глобальной разделяемой памяти (метка shared для переменных)
- Приватные данные доступны только одному потоку (метка private для переменных)



По умолчанию

- Все переменные shared общие, кроме:
  - Индексов параллельных циклов
  - Переменных, объявленных внутри параллельных регионов

Уровни параллельности

- SPMD
- MPMD
- MDMD

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp target teams distribute parallel for map(from:sinTable[0:256])
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

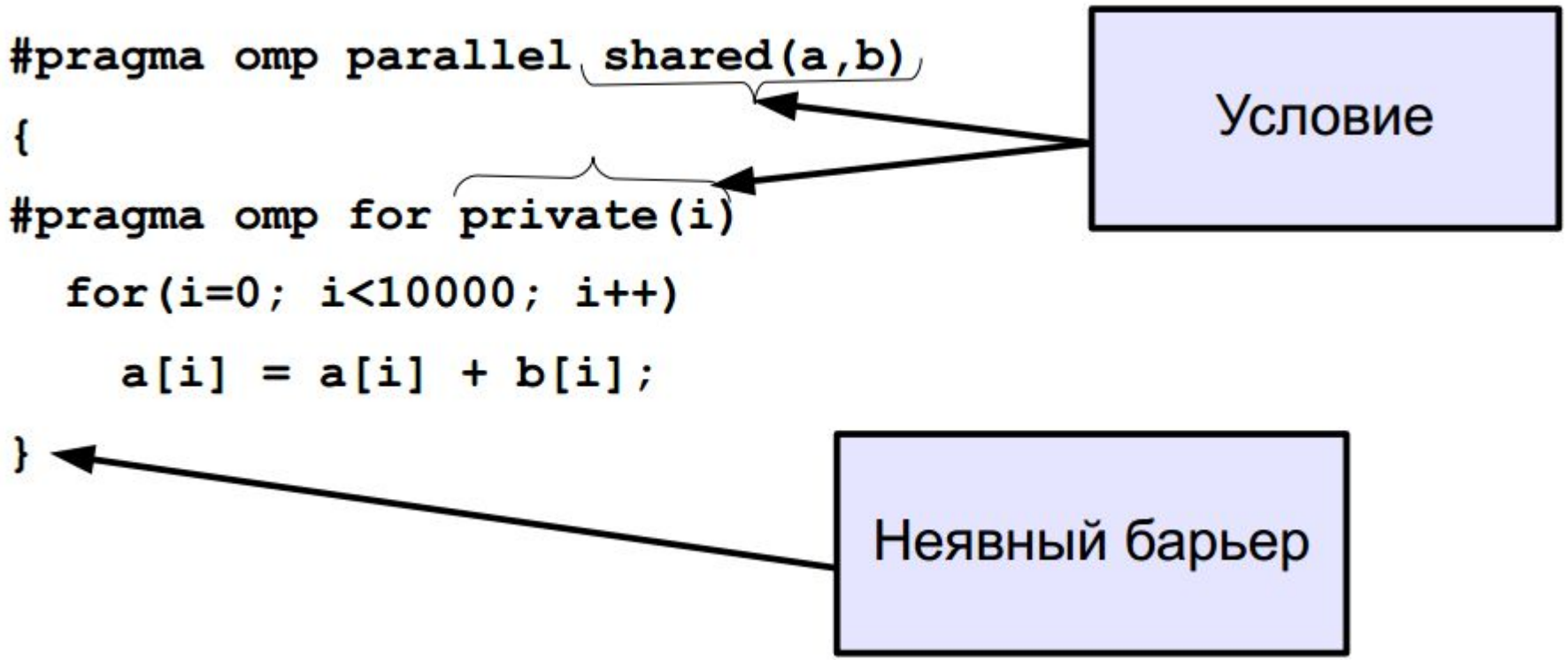
Parallel

- Начинает параллельное выполнение блока в {}
- Создает “команду” потоков
- Количество потоков чаще всего равняется количеству процессоров
- После окончания блока количество потоков становится 1

```
#pragma omp parallel
{
  // Code inside this region runs in parallel.
  printf("Hello!\n");
}
```

Параллельный регион

- Параллельный регион в OpenMP – блок кода, который исполняется всеми потоками параллельно.



Условный параллелизм

- Если условие равно 0 параллельность не работает

```
extern int parallelism_enabled;
#pragma omp parallel for if(parallelism_enabled)
for(int c=0; c<n; ++c)
  handle(c);
```

For

- Без pragma parallel выполняется в один поток

```
#pragma omp for
for(int n=0; n<10; ++n)
{
  printf(" %d", n);
}
printf(".\n");
```

```
#pragma omp parallel
{
 #pragma omp for
 for(int n=0; n<10; ++n) printf(" %d", n);
}
printf(".\n");
```

```
#pragma omp parallel for
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

Parallel for

- С pragma parallel выполняется в текущей «команде» потоков

```
#pragma omp parallel
{
 #pragma omp for
 for(int n=0; n<10; ++n) printf(" %d", n);
}
printf(".\n");
```

```
#pragma omp parallel for
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

Num_threads(N)

- Определяет количество потоков в «команде»

```
#pragma omp parallel num_threads(3)
{
  // This code will be executed by three threads.

  // Chunks of this loop will be divided amongst
  // the (three) threads of the current team.
  #pragma omp for
  for(int n=0; n<10; ++n) printf(" %d", n);
}
```

Циклы, которые нельзя распараллелить

- Рекурсивные зависимости

```
for (int i = 1; i <= n; ++i)
  a[i] = a[i-1] + b[i];
```

- Например, числа Фибоначчи

Зависимости

- Решение зависимостей:
  - перегруппировка кода
  - переписывание алгоритмов
  - расстановка барьеров
- Если зависимости есть, но мы с ними ничего не сделали, компилятор не будет спорить, а программа будет работать неверно

Sheduling

- Static
- Dynamic
- Guided
- Auto

```
#pragma omp for schedule(static)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

```
#pragma omp for schedule(dynamic)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

Ordered

- Код внутри ordered выполняется в установленном циклом последовательности

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

  #pragma omp ordered
  send(files[n]);
}
```

collapse

- Используется для вложенных циклов

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
```

Sections

- Определяют, что должно быть параллельным
- Work1, Work2 + Work3 and Work4 – выполняются паралелльно, но Work2 + Work3 выполняются последовательно

```
#pragma omp sections
{
  { Work1(); }
  #pragma omp section
  { Work2();
    Work3(); }
  #pragma omp section
  { Work4(); }
}
```

```
#pragma omp parallel // starts a new team
{
  //Work0(); // this function would be run by all threads.

  #pragma omp sections // divides the team into sections
  {
    // everything herein is run only once.
    { Work1(); }
    #pragma omp section
    { Work2();
      Work3(); }
    #pragma omp section
    { Work4(); }
  }

  //Work5(); // this function would be run by all threads.
}
```

Atomic

- Определяет атомарную операцию

```
#pragma omp atomic
counter += value;
```

```
#pragma omp atomic update // The word "update" is optional
// One of these:
++x; --x; x++; x--;
x += expr;  x -= expr;  x *= expr;   x /= expr;  x &= expr;
x = x+expr; x = x-expr; x = x*expr;  x = x/expr; x = x&expr;
x = expr+x; x = expr-x; x = expr*x;  x = expr/x; x = expr&x;
x |= expr;  x ^= expr;  x <<= expr;  x >>= expr;
x = x|expr; x = x^expr; x = x<<expr; x = x>>expr;
x = expr|x; x = expr^x; x = expr<<x; x = expr>>x;
```

critical

- Гарантирует выполнение участка кода только одним потоком

```
#pragma omp critical(dataupdate)
{
  datastructure.reorganize();
}
...
#pragma omp critical(dataupdate)
{
  datastructure.reorganize_again();
}
```

Типы переменных

- Private –копия переменной
- shared

```
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
  #pragma omp atomic
  b += a;
}
```

Private, firstprivate

- K

```
#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel private(a,c) shared(b) num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }
}
```

- kx

```
#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel firstprivate(a,c) shared(b) num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }
}
```

reduction

- Совмещает private, shared, и atomic
- Вначале блока копирует значение shared переменной
- В конце блока возвращается в переменную указанным оператором

```
int factorial(int number)
{
  int fac = 1;
  #pragma omp parallel for reduction(*:fac)
  for(int n=2; n<=number; ++n)
    fac *= n;
  return fac;
}
```

- http://bisqwit.iki.fi/story/howto/openmp