Параллельное программирование и суперкомпьютерный кодизайн

Смирнов А.В. asmirnov@srcc.msu.ru

Раздел 5. Многопоточное программирование на основе OpenMP

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- Введение. Компиляция программ, базовые вызовы
- Параллелизация цикла на OpenMP
- Область видимости переменных
- OpenMP и SIMD
- Более сложная параллелизация - задания в OpenMP и вложенный параллелизм

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- **Введение. Компиляция программ, базовые вызовы**
- Параллелизация цикла на OpenMP
- Область видимости переменных
- OpenMP и SIMD
- Более сложная параллелизация - задания в OpenMP и вложенный параллелизм

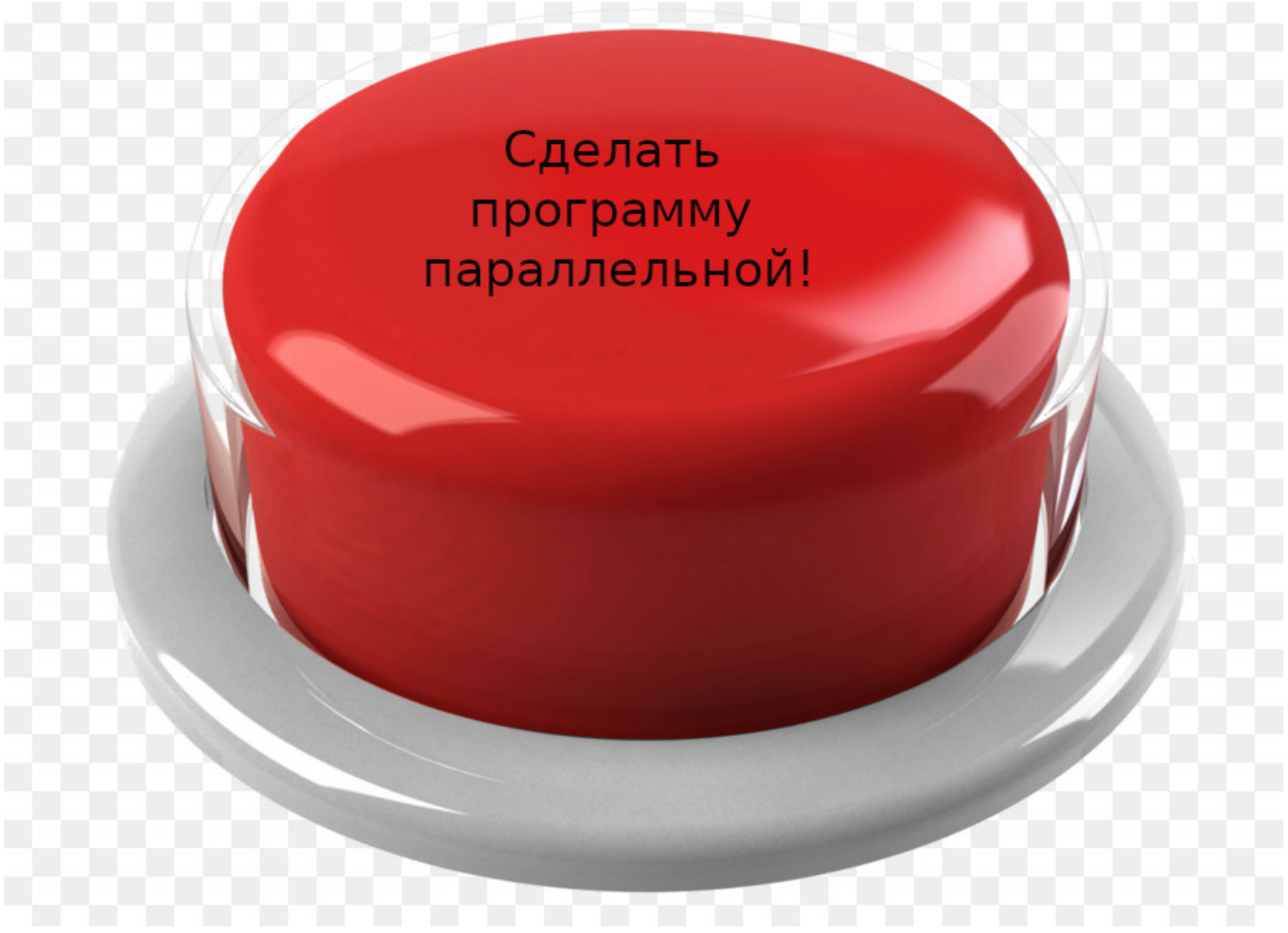
Сделать программу параллельной

- В предыдущем разделе мы разобрались с тем, как программировать в общей памяти с использованием потоков с целью параллелизации программы

Сделать программу параллельной

- В предыдущем разделе мы разобрались с тем, как программировать в общей памяти с использованием потоков с целью параллелизации программы
- Можно ли как-то проще? Хочу волшебную кнопку!

Сделать программу параллельной



Сделать программу параллельной

- В предыдущем разделе мы разобрались с тем, как программировать в общей памяти с использованием потоков с целью параллелизации программы
- Можно ли как-то проще? Хочу волшебную кнопку!
- OpenMP – это максимальное приближение к волшебной кнопке по созданию параллельных программ

Использование OpenMP

- OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций.

Использование OpenMP

- OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций.
- Использование на C++ начинается с подключения библиотеки `#include <omp.h>`, представляющей ряд определенных в этом заголовочном файле функций

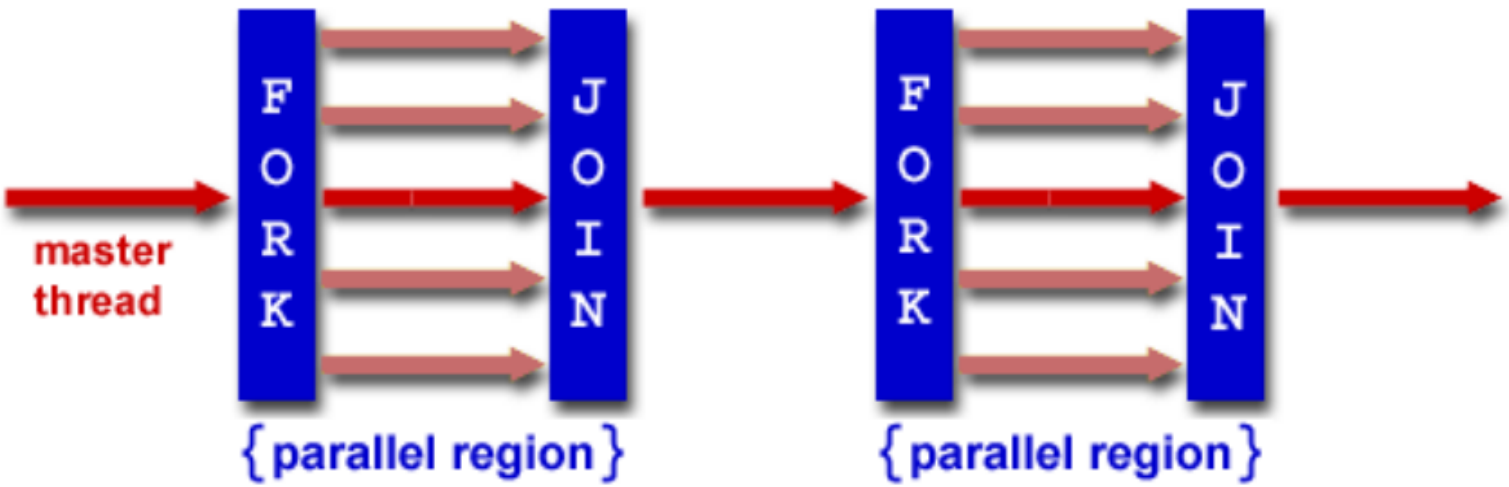
Использование OpenMP

- OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций.
- Использование на C++ начинается с подключения библиотеки `#include <omp.h>`, представляющей ряд определенных в этом заголовочном файле функций
- Также ключевым моментом являются макросы, директивы препроцессора типа `#pragma omp parallel for`

Использование OpenMP

- OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций.
- Использование на C++ начинается с подключения библиотеки `#include <omp.h>`, представляющей ряд определенных в этом заголовочном файле функций
- Также ключевым моментом являются макросы, директивы препроцессора типа `#pragma omp parallel for`
- Для сборки требуется добавление `-fopenmp` и на этапе компиляции (для того, чтобы препроцессор правильно обработал эти директивы) и на этапе линковки, чтобы задействовать требуемые библиотеки

Модель выполнения OpenMP приложения



Простейшая программа с OpenMP

```
#include <iostream>
#include <omp.h>
void hello()
{
    std::cout<<"Hello Other World"<<std::endl;
}
int main()
{
#pragma omp parallel
    hello();
}
```

Простейшая программа с OpenMP

```
#include <iostream>
#include <omp.h>
void hello()
{
    std::cout<<"Hello Other World"<<std::endl;
}
int main()
{
#pragma omp parallel
    hello();
}
```

- Компиляция: g++ -o name name.cpp -fopenmp

Определение числа потоков

Сколько раз напечатается приветствие?

Определение числа потоков

Сколько раз напечатается приветствие?

- По умолчанию – столько раз, сколько решит библиотека

Определение числа потоков

Сколько раз напечатается приветствие?

- По умолчанию – столько раз, сколько решит библиотека
- Можно это контролировать через переменную окружения `OMP_NUM_THREADS`

Определение числа потоков

Сколько раз напечатается приветствие?

- По умолчанию – столько раз, сколько решит библиотека
- Можно это контролировать через переменную окружения `OMP_NUM_THREADS`
- Или лучше из программы функцией `omp_set_num_threads()`

Определение числа потоков

Сколько раз напечатается приветствие?

- По умолчанию – столько раз, сколько решит библиотека
- Можно это контролировать через переменную окружения `OMP_NUM_THREADS`
- Или лучше из программы функцией `omp_set_num_threads()`
- Можно предварительно определить число процессов через `omp_get_num_procs()` (обычно учитывает гипертрединг)

Определение числа потоков

- В некоторых случаях целесообразно устанавливать число тредов динамически в зависимости от загрузки имеющихся процессоров.
- Можно использовать функцию `omp_set_dynamic(int flag)` (но, если честно, мне не удалось добиться того, чтобы на ноутбуке она задействовала меньше ядер; поэтому используйте с осторожностью)

Определение своего места в мире

- `int omp_get_num_procs()` возвращает количество процессоров в системе;
- `int omp_get_num_threads()` возвращает количество тредов, выполняющих параллельный участок (меняется только на последовательных участках);
- `int omp_get_thread_num()` возвращает номер вызывающего треда.

Вывод номера потока

```
#include <iostream>
#include <omp.h>
void hello()
{
    std::cout<<"Hello from "
        <<omp_get_thread_num()<<std::endl;
}
int main()
{
#pragma omp parallel
    hello();
}
```

Вывод номера потока

```
Hello from Hello from 0Hello from 6
Hello from Hello from 1Hello from 5
Hello from 3
7
4
Hello from 2
```

- Что это за каша у меня получилась?

Вывод номера потока

```
#include <iostream>
#include <omp.h>
void hello()
{
#pragma omp critical
    std::cout<<"Hello from "
        <<omp_get_thread_num()<<std::endl;
}
int main()
{
#pragma omp parallel
    hello();
}
```

Вывод номера потока

```
Hello from 0
Hello from 6
Hello from 5
Hello from 2
Hello from 3
Hello from 1
Hello from 4
Hello from 7
```

- Критическая секция заставляет вывод не происходить одновременно. Но порядком мы, естественно, не управляем.

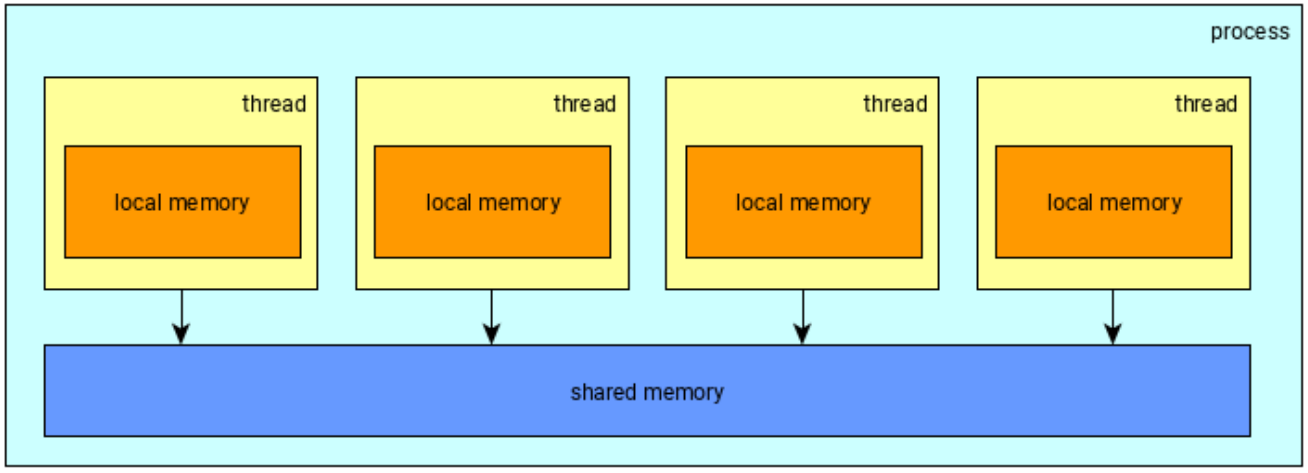
Вывод номера потока

Как это вообще работает?

- Директива `#pragma omp parallel` при опции `-fopenmp` превращается препроцессом в C++ код – OpenMP запускает потоки

Как это вообще работает?

- Директива `#pragma omp parallel` при опции `-fopenmp` превращается препроцессом в C++ код – OpenMP запускает потоки



Как это вообще работает?

- Директива `#pragma omp parallel` при опции `-fopenmp` превращается препроцессом в C++ код – OpenMP запускает потоки
- Эти потоки и обращаются к общей памяти и имеют свою память (например, в которой хранится номер потока).

Как это вообще работает?

- Директива `#pragma omp parallel` при опции `-fopenmp` превращается препроцессом в C++ код – OpenMP запускает потоки
- Эти потоки и обращаются к общей памяти и имеют свою память (например, в которой хранится номер потока).
- По окончании общей секции потоки объединяются

Как это вообще работает?

- Директива `#pragma omp parallel` при опции `-fopenmp` превращается препроцессом в C++ код – OpenMP запускает потоки
- Эти потоки и обращаются к общей памяти и имеют свою память (например, в которой хранится номер потока).
- По окончании общей секции потоки объединяются
- В отличии от использования библиотеки `threads` у нас нет как такового вызова какой-то функции, поэтому и нет таких понятий как передать параметр в OpenMP или получить возвращаемое значение. Но зная номер своего потока, OpenMP потоки могут записывать что-то в различные ячейки памяти, зависящие от этого номера

Вопросы?

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- Введение. Компиляция программ, базовые вызовы
- **Параллелизация цикла на OpenMP**
- Область видимости переменных
- OpenMP и SIMD
- Более сложная параллелизация - задания в OpenMP и вложенный параллелизм

Потоки, потоки, а где же параллельность?

Давайте реализуем простой пример сложения двух векторов в параллельном режиме

Напомним пример по сложению векторов

```
std::random_device rd; std::mt19937 gen(rd());
std::uniform_real_distribution<double>
    uid(0., 1.);
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a (n);
std::vector<double> b (n);
std::vector<double> c (n);
std::generate(a.begin(),a.end(),
    [&uid,&gen]()->double {return uid(gen);}
);
std::generate(b.begin(),b.end(),
    [&uid,&gen]()->double {return uid(gen);}
);
for (size_t i = 0; i != n; ++i) {
    c[i] = a[i] + b[i];
}
```

Реализация на OpenMP

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    // vector declaration and generation here
    #pragma omp parallel for
    for (size_t i = 0; i != n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Вывод номера потока

Реализация на OpenMP

- Да, а еще можно задать количество потоков

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    // vector declaration and generation here
    #pragma omp parallel for num_threads(4)
    for (size_t i = 0; i != n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Реализация на OpenMP

```
#include <chrono>
int main() {
  constexpr size_t n = 32 * 1024 * 1024;
  // vector declaration and generation here
  auto start=std::chrono::steady_clock::now();
  #pragma omp parallel for num_threads(4)
  for (size_t i = 0; i != n; ++i) {
    c[i] = a[i] + b[i];
  }
  auto end = std::chrono::steady_clock::now();
  int time = std::chrono::duration_cast<
    std::chrono::milliseconds
  >(diff).count();
  std::cout << time << " ms "<< std::endl;
}
```

Реализация на OpenMP

Пример тестовый, большого ускорения не будет – мы упираемся в скорость работы памяти. Нужен другой пример.

Планирование итераций: schedule

- OpenMP позволяет управлять распределением итераций цикла между потоками с помощью директивы `schedule`.
- Основные типы:
  - `static` — заранее делим итерации на блоки фиксированного размера
  - `dynamic` — поток берет новую порцию после завершения предыдущей
  - `guided` — блоки уменьшаются со временем (динамическое с уменьшением блока)
- Можно дополнительно указать размер блока: `schedule(type, chunk)`

Пример: разные затраты на итерации

```
int main() {
  constexpr size_t n = 16;
  std::vector<int> a(n);
  #pragma omp parallel for schedule(static, 1)
  for (size_t i = 0; i < n; ++i) {
    // different complexity
    std::this_thread::sleep_for(
      std::chrono::milliseconds(i*10)
    );
    a[i] = i*i;
    printf("Thread %d processed iteration %zu\n",
       omp_get_thread_num(), i);
  }
}
```

Всегда один и тот же вывод

```
Thread 0 processed iteration 0
Thread 1 processed iteration 1
Thread 2 processed iteration 2
Thread 3 processed iteration 3
Thread 0 processed iteration 4
Thread 1 processed iteration 5
Thread 2 processed iteration 6
Thread 3 processed iteration 7
Thread 0 processed iteration 8
Thread 1 processed iteration 9
Thread 2 processed iteration 10
Thread 3 processed iteration 11
Thread 0 processed iteration 12
Thread 1 processed iteration 13
Thread 2 processed iteration 14
Thread 3 processed iteration 15
```

Пример: разные затраты на итерации

```
int main() {
  constexpr size_t n = 16;
  std::vector<int> a(n);
  #pragma omp parallel for schedule(dynamic, 1)
  for (size_t i = 0; i < n; ++i) {
    // different complexity
    std::this_thread::sleep_for(
      std::chrono::milliseconds(i*10)
    );
    a[i] = i*i;
    printf("Thread %d processed iteration %zu\n",
       omp_get_thread_num(), i);
  }
}
```

Возможен различный вывод

```
Thread 0 processed iteration 0
Thread 2 processed iteration 1
Thread 1 processed iteration 2
Thread 3 processed iteration 3
Thread 0 processed iteration 4
Thread 2 processed iteration 5
Thread 1 processed iteration 6
Thread 3 processed iteration 7
Thread 0 processed iteration 8
Thread 2 processed iteration 9
Thread 1 processed iteration 10
Thread 3 processed iteration 11
Thread 0 processed iteration 12
Thread 2 processed iteration 13
Thread 1 processed iteration 14
Thread 3 processed iteration 15
```

Dynamic vs Guided (с указанием размера блока - chu

Основные различия

Schedule	Размер блока	Как распределяется
dynamic	фиксированный (chunk)	Поток берёт блок размером chunk, когда освобождается
guided	уменьшающийся	Поток берёт большие блоки сначала, уменьшаются до chunk

- dynamic — блоки всегда одинаковые (кроме последнего)
- guided — блоки сначала большие, уменьшаются до chunk

Вопросы?

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- Введение. Компиляция программ, базовые вызовы
- Параллелизация цикла на OpenMP
- **Область видимости переменных**
- OpenMP и SIMD
- Более сложная параллелизация - задания в OpenMP и вложенный параллелизм

Области видимости переменных

- В OpenMP каждая переменная может быть:
  - `shared` — общая для всех потоков
  - `private` — локальная для каждого потока
  - `firstprivate` / `lastprivate`
  - `reduction` — слияние частичных результатов
- Явное указание областей видимости снижает риск гонок данных

shared: общие переменные

- Общая копия переменной для всех потоков
- Несинхронизированная запись ведёт к гонкам данных

```
int sum = 0;
#pragma omp parallel for shared(sum)
for (int i = 0; i < 100; ++i) {
    sum += i; // data race
}
```

private: локальные переменные

- У каждого потока — своя копия переменной
- Начальное значение не копируется

```
int x = 5;
#pragma omp parallel private(x)
{
    x = omp_get_thread_num();
    printf("Thread %d: x = %d\n",
           omp_get_thread_num(), x);
}
printf("After: x = %d\n", x);
```

firstprivate: копия с инициализацией

- Каждому потоку создаётся своя копия переменной
- Начальное значение копируется из оригинала
- Используется, если значение нужно внутри потока, но менять его можно независимо

```
int base = 10;
#pragma omp parallel firstprivate(base)
{
    base += omp_get_thread_num();
    printf("Thread %d: base = %d\n",
           omp_get_thread_num(), base);
}
```

✓ Все потоки начинают с одинакового значения

✗ Изменения не сохраняются после выхода из блока

lastprivate: сохранить результат

- Применяется к циклам (`for`, `sections`)
- После завершения блока значение переменной из последней итерации сохраняется в оригинал
- Удобно, если результат последней итерации нужен вне параллельной области

```
int val = 0;
#pragma omp parallel for lastprivate(val)
for (int i = 0; i < 5; ++i) {
    val = i;
}
printf("After parallel: val = %d\n", val); // 4
```

✓ Сохраняется значение из последней итерации

✗ Не предотвращает гонки, если есть общие переменные

Попробуем сложить все элементы массива

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp parallel for
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

Попробуем сложить все элементы массива

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp parallel for
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

- Без OpenMP 222ms, с OpenMP... странные результаты

Опция reduction

- Опция reduction определяет что на выходе из параллельного блока переменная получит комбинированное значение.
  Пример:
- `#pragma omp for reduction(+ : x)`
- Допустимы следующие операции:

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

Опция reduction

- Опция reduction определяет значение переменных, входящих в список ее аргументов, на главном потоке после завершения параллельного участка как результат выполнения редуктивной операции. На каждом из потоков, выполняющих параллельный участок, переменная инициализируется значением, соответствующим редуктивной операции.
- Пусть параллельный участок выполнялся n потоками, и до него переменная имела значение v. Если в конце выполнения параллельного участка локальные копии переменной a имели значения v 1 , . . . , v n , то после параллельного участка переменная a на главном потоке получит значение, равное (v x v 1 x v 2 x . . . x v n).

Попробуем сложить все элементы массива

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp parallel for reduction(+:res)
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

- Теперь программа написана правильно, и ускорение есть. Замеряем результат

Проведем сравнение

AMD Ryzen 7 3750H (4 core)

- Без OpenMP: 222 ms

Проведем сравнение

AMD Ryzen 7 3750H (4 core)

- Без OpenMP: 222 ms
- `num_threads(2)`: 139 ms

Проведем сравнение

AMD Ryzen 7 3750H (4 core)

- Без OpenMP: 222 ms
- `num_threads(2)`: 139 ms
- `num_threads(4)`: 98 ms

Проведем сравнение

AMD Ryzen 7 3750H (4 core)

- Без OpenMP: 222 ms
- `num_threads(2)`: 139 ms
- `num_threads(4)`: 98 ms
- `num_threads(8)`: 78 ms (даже перебили async с продвинутой техникой на 8 ядрах, хотя на 4 он как раз выдает 78)

А какие еще были проблемы у кода без reduction?

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp parallel
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

А какие еще были проблемы у кода без reduction?

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp parallel
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

- Непредсказуемый результат, вообще говоря не являющийся суммой
- Проверим это на счетчике

Проверяем работу счетчика

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    int res = 0.;
    #pragma omp parallel for num_threads(8)
    for (size_t i = 0; i != n; ++i) {
        res++;
    }
    std::cout<<res<<std::endl;
}
```

Проверяем работу счетчика

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    int res = 0.;
    #pragma omp parallel for num_threads(8)
    for (size_t i = 0; i != n; ++i) {
        res++;
    }
    std::cout<<res<<std::endl;
}
```

- 4669451, 4615065,... что за результаты?

Проверяем работу счетчика

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    int res = 0.;
    #pragma omp parallel for num_threads(8)
    for (size_t i = 0; i != n; ++i) {
        res++;
    }
    std::cout<<res<<std::endl;
}
```

- 4669451, 4615065,... что за результаты?
- Что делать, если все же нужен счетчик?

Реализуем счетчик

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    int res = 0.;
    #pragma omp parallel for num_threads(8)
    for (size_t i = 0; i != n; ++i) {
    #pragma omp atomic
        res++;
    }
    std::cout<<res<<std::endl;
}
```

Реализуем счетчик

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    int res = 0.;
    #pragma omp parallel for num_threads(8)
    for (size_t i = 0; i != n; ++i) {
    #pragma omp atomic
        res++;
    }
    std::cout<<res<<std::endl;
}
```

- Теперь всегда 33554432

Реализуем счетчик

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    int res = 0.;
    #pragma omp parallel for num_threads(8)
    for (size_t i = 0; i != n; ++i) {
    #pragma omp atomic
        res++;
    }
    std::cout<<res<<std::endl;
}
```

- Теперь всегда 33554432
- Надо понимать, что это работает относительно медленно

OpenMP: Critical

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    #pragma omp critical
    sum += i;
}
```

Особенности:

- Гарантирует, что блок выполняет один поток.
- Универсально, для любых операций.
- Может быть медленным при большом числе потоков.

Сравнение с C++:

OpenMP	C++ std
`#pragma omp critical`	`std::mutex`

OpenMP: Atomic

```
int count = 0;
#pragma omp parallel for
for (int i = 0; i < N; i++)
    #pragma omp atomic
    count++;
```

Особенности:

- Для простых операций (+, -, ++, –, &=, |= и т.д.)
- Быстрее, чем `critical`.
- Минимизирует блокировки потоков.

Сравнение с C++:

OpenMP	C++ std
`#pragma omp atomic`	`std::atomic<int>`

Data scoping по умолчанию

- OpenMP может автоматически определять области видимости переменных
- `default(shared)` — все переменные общие по умолчанию
- `default(none)` — все переменные требуют явного указания (`shared` или `private`)
- Рекомендуется использовать `default(none)` для надёжности и читаемости кода

```
#pragma omp parallel default(none) \
    shared(a,b,c) private(i)
{
    for(int i = 0; i < 100; ++i)
        c[i] = a[i] + b[i];
}
```

✓ Явное управление областями видимости

✗ Неопределённые переменные вызывают ошибки компиляции

Жизненный цикл переменных в OpenMP

- `shared` — живут столько же, сколько и исходная переменная
- `private / firstprivate / lastprivate` — создаются при входе в параллельную область
- Локальные переменные внутри блока `parallel` существуют только в течение блока
- После выхода из блока `private` переменные уничтожаются, `lastprivate` сохраняет значение при необходимости

Тип переменной	Жизненный цикл
shared	Как оригинал, вне и внутри блока
private	Только внутри блока
firstprivate	Только внутри блока, копия начального значения
lastprivate	Внутри блока + сохранение значение

Типичные ошибки и решения

- ✗ Data race при `shared` без синхронизации
- ✗ Потеря значений при `private` без `lastprivate`
- ✗ Избыточные `critical` — тормозят код
- ✓ Используйте `default(none)` для явного контроля

```
#pragma omp parallel default(none) \
    shared(a, b, c, n) private(i)
{
    for (i = 0; i < n; ++i)
        c[i] = a[i] + b[i];
}
```

Вопросы?

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- Введение. Компиляция программ, базовые вызовы
- Параллелизация цикла на OpenMP
- Область видимости переменных
- **OpenMP и SIMD**
- Более сложная параллелизация - задания в OpenMP и вложенный параллелизм

OpenMP и SIMD

Как мы обсуждали, для одного узла есть два основных типа параллелизма

- По задачам. В данном случае используются все ядра нашего процессора. Идея заключается в выделении независимых задач, которые одновременно могут выполняться на нескольких ядрах.

OpenMP и SIMD

Как мы обсуждали, для одного узла есть два основных типа параллелизма

- По задачам. В данном случае используются все ядра нашего процессора. Идея заключается в выделении независимых задач, которые одновременно могут выполняться на нескольких ядрах.
- По данным (векторизация): используются регистры увеличенной длины каждого ядра. В этом случае происходит обработка сразу нескольких элементов (векторов) данных за одну операцию/инструкцию, отсюда и название «векторизация».

OpenMP и SIMD

Как мы обсуждали, для одного узла есть два основных типа параллелизма

- По задачам. В данном случае используются все ядра нашего процессора. Идея заключается в выделении независимых задач, которые одновременно могут выполняться на нескольких ядрах.
- По данным (векторизация): используются регистры увеличенной длины каждого ядра. В этом случае происходит обработка сразу нескольких элементов (векторов) данных за одну операцию/инструкцию, отсюда и название «векторизация».

Начиная с версии 4.0 OpenMP также предоставляет и параллелизм по данным. Попробуем им воспользоваться.

OpenMP и SIMD

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp simd reduction(+:res)
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

Как думаете, будет эффект?

OpenMP и SIMD

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp simd reduction(+:res)
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

Как думаете, будет эффект?

Нет, совершенно никакого... а почему

OpenMP и SIMD

```
int main() {
    constexpr size_t n = 256 * 1024 * 1024;
    double res = 0.;
    // vector declaration and generation here
    #pragma omp simd reduction(+:res)
    for (size_t i = 0; i != n; ++i) {
        res += a[i];
    }
}
```

Как думаете, будет эффект?
Нет, совершенно никакого... а почему

- OpenMP боится применять SIMD инструкции, поскольку не знает ничего о выравнивании данных. И вообще в цикл передан вектор, а си-код из OpenMP плохо это понимает.

Перепишем код с указателями и информации о выравнивании

```
constexpr size_t n = 256 * 1024 * 1024;
double res = 0.;
// vector declaration and generation here
double* ap = a.data();
#pragma omp simd aligned(ap: 32)
    reduction(+:res)
for (size_t i = 0; i != n; ++i) {
    res += ap[i];
}
```

Перепишем код с указателями и информации о выравнивании

```
constexpr size_t n = 256 * 1024 * 1024;
double res = 0.;
// vector declaration and generation here
double* ap = a.data();
#pragma omp simd aligned(ap: 32)
    reduction(+:res)
for (size_t i = 0; i != n; ++i) {
    res += ap[i];
}
```

- У нас ускорение. 135ms вместо 222. Ровно как было, когда мы писали при помощи интринсиков sse на прошлой лекции. А можно это улучшить до avx и как?

OpenMP и SIMD

- Компилируем с -mavx... и программа падает. Почему?

OpenMP и SIMD

- Компилируем с -mavx... и программа падает. Почему?
- Необходимо действительно выравнять вектор, как это мы делали в теме про интринскики. В результате получается очередное ускорение, 109ms, ровно как было с интринсиками. Возможно, их использовать напрямую и не нужно?)

OpenMP и SIMD

- Компилируем с -mavx... и программа падает. Почему?
- Необходимо действительно выравнять вектор, как это мы делали в теме про интринскики. В результате получается очередное ускорение, 109ms, ровно как было с интринсиками. Возможно, их использовать напрямую и не нужно?)
- Но что хорошо в OpemMP это возможность объединить подходы и использовать одновременно параллелизм по задачам и по данным!

Включаем оба вида паралеллизма

```
constexpr size_t n = 256 * 1024 * 1024;
double res = 0.;
// vector declaration and generation here
double* ap = a.data();
#pragma omp parallel for simd num_threads(4)
        aligned(ap: 32) reduction(+:res)
for (size_t i = 0; i != n; ++i) {
    res += ap[i];
}
```

Включаем оба вида паралеллизма

- Simd + avx + num_threads(2): 81ms

OpenMP и SIMD

Включаем оба вида паралеллизма

- Simd + avx + num_threads(2): 81ms
- Simd + avx + num_threads(4): 75ms. Новый рекорд!

OpenMP и SIMD

Включаем оба вида паралеллизма

- Simd + avx + num_threads(2): 81ms
- Simd + avx + num_threads(4): 75ms. Новый рекорд!
- Simd + avx + num_threads(8): 77ms. Чуть хуже

OpenMP и SIMD

Включаем оба вида паралеллизма

- Simd + avx + num_threads(2): 81ms
- Simd + avx + num_threads(4): 75ms. Новый рекорд!
- Simd + avx + num_threads(8): 77ms. Чуть хуже
- Правда, переписав async + интринсики я получил те же числа

Далее сводная таблица по всем способам

Сводная таблица по суммированию вектора

Мы просуммировали вектор разными способами. Попробуем порассуждать и сделать выводы

linear: 222
async with 2-4-8 threads: 140-103-87
reduce seq: 130
reduce par: 82
async with code from reduce and 2-4-8 threads: 100-78-82
openmp with parallel num_threads(2-4-8) 139-98-78
openmp simd (without and with -mavx) 135-109
openmp simd (avx) + parallel 81 - 75 - 77
intrinsic sse and avx: 135-109
intrinsic avx + async with 2-4-8 threads: 81 - 75 - 77

Вопросы?

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- Введение. Компиляция программ, базовые вызовы
- Параллелизация цикла на OpenMP
- Область видимости переменных
- OpenMP и SIMD
- **Более сложная параллелизация - задания в OpenMP и вложенный параллелизм**

Алгоритм сортировки QuickSort

QuickSort – один из наиболее известных и при этом эффективных алгоритмов сортировки

- Опишем этот алгоритм, сначала линейную реализацию
- После этого мы обсудим механизм заданий в OpenMP и перейдем к параллельной реализации

Алгоритм сортировки QuickSort

На входе массив a[0]...a[N-1] и опорный элемент p (обычно средний элемент в массиве), по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.

Алгоритм сортировки QuickSort

На входе массив a[0]...a[N-1] и опорный элемент p (обычно средний элемент в массиве), по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...

Алгоритм сортировки QuickSort

На входе массив a[0]...a[N-1] и опорный элемент p (обычно средний элемент в массиве), по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...
4. Повторяем шаг 3, пока i <= j

Алгоритм сортировки QuickSort

На входе массив a[0]...a[N-1] и опорный элемент p (обычно средний элемент в массиве), по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...
4. Повторяем шаг 3, пока i <= j
5. Выполняем всю процедуру рекурсивно для левой и правой части

Алгоритм сортировки QuickSort

Последовательная реализация

```
void quickSort(std::vector<double>& v,
const size_t start, const size_t end) {
    size_t i = start, j = end;
    double pivot = v[(start + end)/2];
    do {
        while (v[i] < pivot) ++i;
        while (v[j] > pivot) --j;
        if (i <= j) {
            std::swap(v[i], v[j]);
            ++i;
        }
    } while (i <= j);
    if (j > start) quickSort(v, start, j);
    if (i < end) quickSort(v, i, end);
}
```

Последовательная реализация

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    std::vector<double> a (n);
    // fill vector here
    quickSort(a, 0, n - 1);
}
```

- У нас рекурсия (причем оправданная), и схема с параллелизацией цикла не поможет. Нужен механизм заданий.

Механизм заданий в OpenMP

`#pragma omp task`

- Генерируется задание для структурного блока. Поток, который выполняет конструкцию task либо сразу выполняет задание, либо откладывает выполнение – тогда задание может быть выполнено любым потоком из группы.
- Если накладные расходы по организацию выполнения задания слишком велики, task будет выполнен пришедшим к инструкции потоком.

Механизм заданий в OpenMP

`#pragma omp single`

- Директива single определяет что последующий блок будет выполняться только одним тредом
- Директива зачастую используется для создания заданий одним потоком внутри параллельной секции

Механизм заданий в OpenMP

```
#pragma omp single {
    for (i = 0; i < ONEZILLION; i++)
        #pragma omp task
        process(items[i]);
}
```

- В какой-то момент, когда список отложенных задачи слишком длинный, может быть остановлена генерация новых задач, и каждый поток занимается выполнением уже сгенерированных задач

Механизм заданий в OpenMP

`#pragma omp taskwait`

- Используется для синхронизации заданий, созданных в текущем участке (тот же уровень вложенности) до директивы taskwait.

Механизм заданий в OpenMP

```
#pragma omp single {
    for (i = 0; i < ONEZILLION; i++)
        #pragma omp task
        process(items[i]);
#pragma omp taskwait
    for (i = ONEZILLION; i<2*ONEZILLION; i++)
        #pragma omp task
        process(items[i]);
}
```

- Первая часть заданий выполнится раньше второй части заданий

Попробуем параллелизовать QuickSort

```
void quickSort(std::vector<double>& v,
const size_t start, const size_t end) {
    size_t i = start, j = end;
    double pivot = v[(start + end)/2];
    do {while (v[i] < pivot) ++i;
        while (v[j] > pivot) --j;
        if (i <= j) {
            std::swap(v[i], v[j]);  ++i;
        }
    } while (i <= j);
#pragma omp task
    if (j > start) quickSort(v, start, j);
#pragma omp task
    if (i < end) quickSort(v, i, end);
#pragma omp taskwait
}
```

Попробуем параллелизовать QuickSort

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    std::vector<double> a (n);
    // fill vector here
#pragma omp parallel num_threads(4)
    {
#pragma omp single
        {
            quickSort(a, 0, n - 1);
        }
    }
}
```

- Мы здесь упускаем что-то важное. Как думаете, что?

Попробуем параллелизовать QuickSort

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    std::vector<double> a (n);
    // fill vector here
#pragma omp parallel num_threads(4)
    {
#pragma omp single
        {
            quickSort(a, 0, n - 1);
        }
    }
}
```

- Мы здесь упускаем что-то важное. Как думаете, что?
- Если запустить такой код, массив не будет полностью отсортирован. Почему?

Области видимости в задачах OpenMP

Проблема

Даже если мы передаём вектор `v` по ссылке, это не гарантирует общий доступ между задачами.

- Переменные функции (`v`, `i`, `j`) — локальные для вызывающего потока.

Области видимости в задачах OpenMP

Проблема

Даже если мы передаём вектор `v` по ссылке, это не гарантирует общий доступ между задачами.

- Переменные функции (`v`, `i`, `j`) — локальные для вызывающего потока.
- При создании `#pragma omp task` OpenMP копирует контекст, включая ссылки.

Области видимости в задачах OpenMP

Проблема

Даже если мы передаём вектор `v` по ссылке, это не гарантирует общий доступ между задачами.

- Переменные функции (`v`, `i`, `j`) — локальные для вызывающего потока.
- При создании `#pragma omp task` OpenMP копирует контекст, включая ссылки.
- То есть каждая задача получает *свою переменную-ссылку*, пусть и указывающую на тот же объект.

Области видимости в задачах OpenMP

Проблема

Даже если мы передаём вектор `v` по ссылке, это не гарантирует общий доступ между задачами.

- Переменные функции (`v`, `i`, `j`) — локальные для вызывающего потока.
- При создании `#pragma omp task` OpenMP копирует контекст, включая ссылки.
- То есть каждая задача получает *свою переменную-ссылку*, пусть и указывающую на тот же объект.
- Такое поведение — `firstprivate` по умолчанию, а не `shared`.

Почему это может быть проблемой

✗ Компилятор не знает, что объект, на который ссылается `v`, живёт дольше всех задач.

Почему это может быть проблемой

- ✗ Компилятор не знает, что объект, на который ссылается `v`, живёт дольше всех задач.
- ✗ Он может считать, что каждая задача имеет *независимый контекст*, и не синхронизировать доступ.

Почему это может быть проблемой

- ✗ Компилятор не знает, что объект, на который ссылается `v`, живёт дольше всех задач.
- ✗ Он может считать, что каждая задача имеет *независимый контекст*, и не синхронизировать доступ.
- ✗ При оптимизациях (LTO, инлайнинг, reordering) часть рекурсий может работать с *копией данных*.

Почему это может быть проблемой

- ✗ Компилятор не знает, что объект, на который ссылается `v`, живёт дольше всех задач.
- ✗ Он может считать, что каждая задача имеет *независимый контекст*, и не синхронизировать доступ.
- ✗ При оптимизациях (LTO, инлайнинг, reordering) часть рекурсий может работать с *копией данных*.
- ✗ Поведение без `shared(v)` — **не определено стандартом OpenMP**.

Почему это может быть проблемой

- ✗ Компилятор не знает, что объект, на который ссылается `v`, живёт дольше всех задач.
- ✗ Он может считать, что каждая задача имеет *независимый контекст*, и не синхронизировать доступ.
- ✗ При оптимизациях (LTO, инлайнинг, reordering) часть рекурсий может работать с *копией данных*.
- ✗ Поведение без `shared(v)` — **не определено стандартом OpenMP**.
- ✓ Явное `shared(v)` гарантирует, что все задачи используют один и тот же контейнер.

Итог: как запомнить

- Переменные внутри функций — локальные, даже если это ссылки.

Итог: как запомнить

- Переменные внутри функций — локальные, даже если это ссылки.
- При создании задач OpenMP копирует контекст (firstprivate по умолчанию).

Итог: как запомнить

- Переменные внутри функций — локальные, даже если это ссылки.
- При создании задач OpenMP копирует контекст (firstprivate по умолчанию).
- Всегда указывайте `shared(v)`, если объект должен быть общим между задачами.

Итог: как запомнить

- Переменные внутри функций — локальные, даже если это ссылки.
- При создании задач OpenMP копирует контекст (firstprivate по умолчанию).
- Всегда указывайте `shared(v)`, если объект должен быть общим между задачами.
- Не полагайтесь на то, что “ссылка и так shared” — это не часть гарантии стандарта.

Главная мысль

`shared` — это не “избыточное уточнение”, а способ зафиксировать поведение при оптимизациях.

Более правильная параллельная реализация

```
void quickSort(std::vector<double>& v,
const size_t start, const size_t end) {
    size_t i = start, j = end;
    double pivot = v[(start + end)/2];
    do {while (v[i] < pivot) ++i;
        while (v[j] > pivot) --j;
        if (i <= j) {
            std::swap(v[i], v[j]);  ++i;
        }
    } while (i <= j);
#pragma omp task shared (v)
    if (j > start) quickSort(v, start, j);
#pragma omp task shared (v)
    if (i < end) quickSort(v, i, end);
#pragma omp taskwait
}
```

Более правильная параллельная реализация

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    std::vector<double> a (n);
    // fill vector here
#pragma omp parallel num_threads(4)
    {
#pragma omp single
        {
            quickSort(a, 0, n - 1);
        }
    }
}
```

- Теперь сортируется правильно, но давайте замеряем производительность

Более правильная параллельная реализация

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    std::vector<double> a (n);
    // fill vector here
#pragma omp parallel num_threads(4)
    {
#pragma omp single
        {
            quickSort(a, 0, n - 1);
        }
    }
}
```

- Теперь сортируется правильно, но давайте замеряем производительность
- Без OpenMP 12306ms, с OpenMP – 31512, при этом процессоры сильно загружены. Почему?

Более правильная параллельная реализация

При разработке заданий стоит учитывать размер этих заданий

- Слишком маленькие задания в конце рекурсии тратят больше времени на запуск и дробление, чем на вычисления!
- Перейдем к финальной реализации – поставим ограничение, чтобы не делать задания для слишком маленьких блоков

Правильная параллельная реализация

```
void quickSort(std::vector<double>& v,
const size_t start, const size_t end) {
    size_t i = start, j = end;
    double pivot = v[(start + end)/2];
    do {while (v[i] < pivot) ++i;
        while (v[j] > pivot) --j;
        if (i <= j) {
            std::swap(v[i], v[j]);  ++i;
        }
    } while (i <= j);
```

- Выше ничего не изменилось

Правильная параллельная реализация

- Отличие в нижней части функции QuickSort

```
    if (end - start <= 128) {
        if (j > start) quickSort(v, start, j);
        if (i < end) quickSort(v, i, end);
        return;
    }
#pragma omp task shared (v)
    if (j > start) quickSort(v, start, j);
#pragma omp task shared (v)
    if (i < end) quickSort(v, i, end);
#pragma omp taskwait
}
```

Правильная параллельная реализация

```
int main() {
    constexpr size_t n = 32 * 1024 * 1024;
    std::vector<double> a (n);
    // fill vector here
#pragma omp parallel num_threads(4)
    {
#pragma omp single
        {
            quickSort(a, 0, n - 1);
        }
    }
}
```

- Число 128 можно регулировать, но остановимся на нем
- Посмотрим теперь результаты замера производительности

Проведем сравнение

AMD Ryzen 7 3750H (8 core)

- Без OpenMP: 12306 ms

Проведем сравнение

AMD Ryzen 7 3750H (8 core)

- Без OpenMP: 12306 ms
- `num_threads(2)`: 7578 ms

Проведем сравнение

AMD Ryzen 7 3750H (8 core)

- Без OpenMP: 12306 ms
- `num_threads(2)`: 7578 ms
- `num_threads(4)`: 5718 ms

Проведем сравнение

AMD Ryzen 7 3750H (8 core)

- Без OpenMP: 12306 ms
- `num_threads(2)`: 7578 ms
- `num_threads(4)`: 5718 ms
- `num_threads(8)`: 4606 ms

Проведем сравнение

AMD Ryzen 7 3750H (8 core)

- Без OpenMP: 12306 ms
- `num_threads(2)`: 7578 ms
- `num_threads(4)`: 5718 ms
- `num_threads(8)`: 4606 ms
- Линейного ускорения нет, но оно вполне приличное. И здесь более заметен эффект от гипертредингa.

Nested Parallelism в OpenMP

- Возможность запускать параллельные регионы внутри других параллельных регионов.
- По умолчанию вложенный параллелизм выключен, то есть если внутри одного параллельного региона (pragma omp parallel) встретится другой параллельный регион, то внутренний pragma omp parallel не создаст новые потоки: `omp_set_nested(0)`
- Для включения: `omp_set_nested(1)` или переменная окружения `OMP_NESTED=TRUE`
- Управление числом потоков во вложенных регионах через `num_threads`.

OpenMP: `sections` и `section`

Позволяет разбить работу между потоками, когда задачи разные, а не итерации одного цикла.

```
#pragma omp parallel sections
{
    #pragma omp section
    taskA();

    #pragma omp section
    taskB();

    #pragma omp section
    taskC();
}
```

- Каждый `section` выполняется отдельным потоком из параллельного региона.
- После завершения всех секций потоки синхронизируются.

Реализация с вложенным параллелизмом

```
void quickSort(std::vector<double>& v,
const size_t start,const size_t end,int depth=0) {
    size_t i = start, j = end;
    double pivot = v[(start + end)/2];
    do {while (v[i] < pivot) ++i;
        while (v[j] > pivot) --j;
        if (i <= j) {
            std::swap(v[i], v[j]);  ++i;
        }
    } while (i <= j);
```

- Появился параметр вложенности

Реализация с вложенным параллелизмом

```
    if (end - start <= 128 || depth > 3) {
        if (j > start) quickSort(v,start,j,depth+1);
        if (i < end) quickSort(v,i,end,depth+1);
        return;
    }
#pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        if (j > start) quickSort(v,start,j,depth+1);
        #pragma omp section
        if (i < end) quickSort(v,i,end,depth+1);
    }
}
```

Сравнение `task` и вложенного параллелизма

Оба подхода позволяют создавать параллельные ветви рекурсивных алгоритмов, но работают по-разному.

Task-based подход

- Задания выполняются в рамках одного параллельного региона.
- Потоки берут задачи из общей очереди.
- Лёгкие накладные расходы на создание задачи.
- Хорошо масштабируется при рекурсии (например, в QuickSort).

Nested parallelism

- Каждый вызов может создавать новый параллельный регион.
- При `omp_set_nested(1)` потоки могут порождать новые пулы потоков.
- Высокие накладные расходы: инициализация потоков, синхронизация.
- Эффективен только при малом числе уровней вложенности.

Сравнение `task` и вложенного параллелизма

Практическое наблюдение (QuickSort)

- Вариант с `task` сортирует быстрее, чем с nested parallelism.
- Замеры показывают: вложенный вариант несколько медленнее из-за дополнительных затрат на управление потоками и меньшей равномерности загрузки потоков.

Общие выводы

OpenMP – удобное современное высокоуровневое средство программирования систем с общей памятью

- Сочетает минимальные изменения в коде программы с высокой производительностью
- Имеется ряд нюансов (задания, уровень доступа к переменным и прочее), которые нужно очень аккуратно учитывать при разработке

Вопросы?

Раздел 5. План.

Многопоточное программирование на основе OpenMP

- Введение. Компиляция программ, базовые вызовы
- Параллелизация цикла на OpenMP
- Область видимости переменных
- OpenMP и SIMD
- Более сложная параллелизация - задания в OpenMP и вложенный параллелизм

Домашнее задание

Внимание! Задание!

1. Найти максимум в массиве
2. Найти количество положительных элементов в массиве
3. Найти сумму массива
4. Найти первое число, превышающее заранее заданное

Вы должны написать программу, которая делает одно из написанного выше. Чтобы определить, что, возьмите остаток от деления вашего номера на 4 и прибавьте 1:

$$(number\%4) + 1$$

Домашнее задание

Внимание! Задание!

Выберите один из методов параллельной реализации:

1. Использование `task` для динамического распределения работы
2. Использование `parallel sections`, где разные секции выполняют разные части работы
3. Использование `parallel for` и `reduction`
4. Использование `atomic` для аккумулирования результата
5. Использование `critical` для защиты общей переменной

Чтобы определить ваш вариант, возьмите целое частное от деления номера на 4, затем остаток от деления на 5 и прибавьте 1:

$$((number/4)\%5) + 1$$

Домашнее задание: тип данных массива

Внимание! Задание!

Варианты чисел в массиве:

1. int (32-bit)
2. float
3. long int / long long int (64-bit)
4. double

Чтобы определить ваш вариант, возьмите целое частное от деления вашего номера на 20 и прибавьте 1:

$$(number/20) + 1$$

Домашнее задание: требования к программе

Внимание! Задание!

- Программа должна быть консольной и работать на любой системе, без специфичных IDE
- Должны быть инструкции по сборке программы (например, Makefile)
- Минимум — команда сборки; желательно — полноценная организация проекта
- Программа должна выводить ваш номер и какую задачу вы решали

Домашнее задание: рекомендации по сборке

Внимание! Задание!

- Makefile или CMake
- Поддержка флага, чтобы можно было включать или отключать параллельную версию, а также указывать количество потоков
- Подача программы единым архивом или через репозиторий
- Комментарим к коду, поясняющие реализацию

Вопросы?