НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Н.И. Лобачевского

~ Нижегородский государственный университет им. Н.И. Лобачевского ~

~ Национальный исследовательский университет ~

~ Институт информационных технологий, математики и механики ~

Параллельное программирование с использованием OpenMP

Сысоев А.В.

к.т.н, доцент каф. МОСТ

Содержание

- Основы
  - Подходы к разработке многопоточных программ
  - Состав OpenMP
  - Модель выполнения
  - Модель памяти
- Создание потоков
  - Формирование параллельной области
  - Задание числа потоков
  - Управление областью видимости
- Библиотека функций
- Привязка потоков

Содержание

- Способы распределения работы между потоками
  - Распараллеливание циклов
  - Распараллеливание циклов с редукцией
  - Векторизация
  - threadprivate-данные
  - Функциональный параллелизм (sections)
  - Механизм задач
- Синхронизация
- Вложенный параллелизм
- Переменные окружения

Основы

Подходы к разработке многопоточных программ

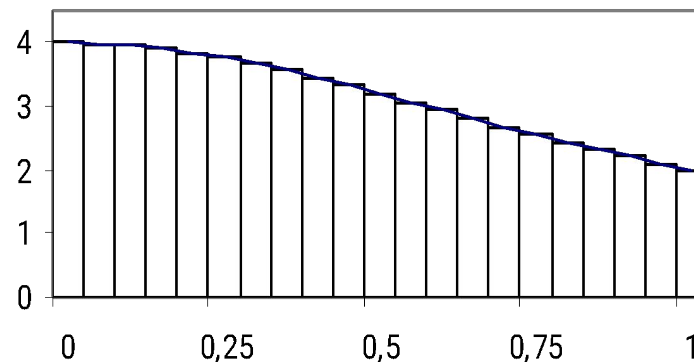
- Использование библиотеки потоков
  (POSIX, Windows Threads, ...)
- Использование возможностей языка программирования
  (C++ 11, …)
- Использование технологии ПП
  (OpenMP, TBB, ...)

Пример: вычисление числа π

- Значение π может быть вычислено через интеграл

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

- Для вычисления определенного интеграла можно воспользоваться методом прямоугольников



Пример программы на C++

```
int num_steps = 100000000;
double step;

double Pi() {
  double x, sum = 0.0;
  step = 1.0 / (double)num_steps;
  for (int i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  return sum;
}

int main() {
  int i;
  double pi = Pi() * step;
return 0;
}
```

Пример программы на Windows Threads…

```
#include <windows.h>
int num_steps = 100000000;
double step;
const int NUM_THREADS = 4;
HANDLE thread_handles[NUM_THREADS];
double global_sum[NUM_THREADS];

void Pi(void *arg) {
  double x, sum = 0.0;
  int start = *(int *)arg;
  step = 1.0 / (double)num_steps;
  for(int i = start; i < num_steps; i = i + NUM_THREADS) {
    x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  global_sum[start] += sum;
}
```

Пример программы на Windows Threads

```
int main() {
  int i;
  double pi = 0;
  DWORD threadID;
  int threadArg[NUM_THREADS];
  for (i = 0; i < NUM_THREADS; i++)
    threadArg[i] = i;
  for (i = 0; i < NUM_THREADS; i++) {
    thread_handles[i] = CreateThread(0, 0,
      (LPTHREAD_START_ROUTINE)Pi, &threadArg[i], 0, &threadID);
  }
  WaitForMultipleObjects(NUM_THREADS, thread_handles,
    TRUE, INFINITE);
  for (i = 0; i < NUM_THREADS; i++)
    pi += global_sum[i] * step;
  return 0;
}
```

Пример программы на C++ Threads…

```
#include <thread>
#include <windows.h>
int num_steps = 100000000;
double step;
const int NUM_THREADS = 4;
double global_sum[NUM_THREADS];

void Pi(int thread_num) {
  double x, sum = 0.0;
  step = 1.0 / (double)num_steps;
  for(int i = thread_num; i < num_steps; i = i + NUM_THREADS) {
    x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  global_sum[thread_num] += sum;
}
```

Пример программы на C++ Threads

```
int main()
{
  int i;
  double pi = 0;
  std::thread threads[NUM_THREADS] =
    { std::thread(Pi, 0), std::thread(Pi, 1),
      std::thread(Pi, 2), std::thread(Pi, 3) };
  for (i = 0; i < NUM_THREADS; i++) {
    threads[i].join();
    pi += global_sum[i] * step;
  }
  printf("%.15lf\n%.15lf\n", PI, pi);
  return 0;
}
```

Пример программы на OpenMP

```
int main()
{
  int num_steps = 100000000;
  const int NUM_THREADS = 4;
  double pi, x, sum = 0.0;
  double step = 1.0 / (double)num_steps;

#pragma omp parallel num_threads(NUM_THREADS) private(x)
#pragma omp for schedule(static, 1) reduction(+:sum)
  for (int i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  pi = sum * step;
  return 0;
}
```

Структура OpenMP

- Компоненты:
  - Набор директив компилятора.

```
#pragma omp имя_директивы [параметр1 ... параметрN]
```

  - Библиотека функций.

```
omp_имя_функции([параметр1, ..., параметрN)
```

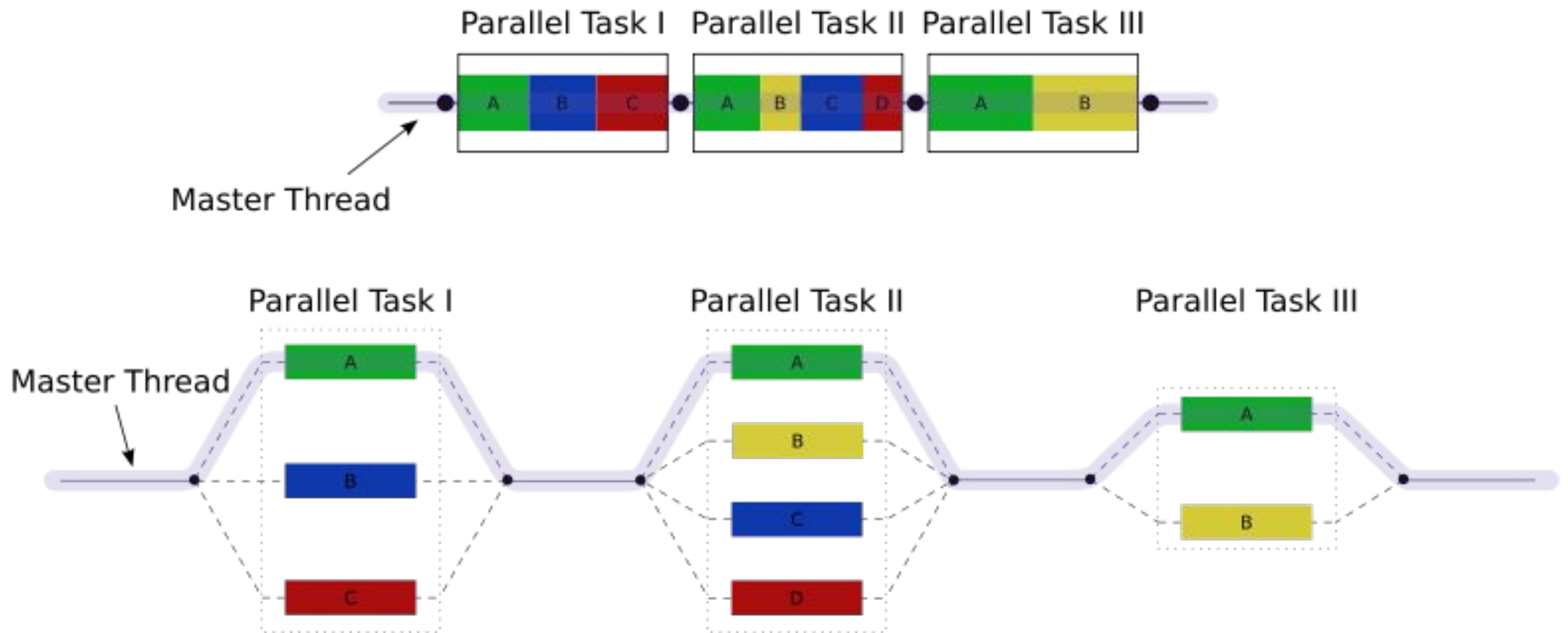
  - Набор переменных окружения.

```
OMP_ИМЯ_ПЕРЕМЕННОЙ
```

Изложение материала будет проводиться на примере языка C/C++.

Модель выполнения

- «Пульсирующий» (fork-join) параллелизм.



*Источник: http://en.wikipedia.org/wiki/OpenMP

Модель памяти

- Рассматривается модель стандартов до OpenMP 4.0 (до появления поддержки гетерогенного программирования)

- В OpenMP-программе два типа памяти: private и shared
- Принадлежность конкретной переменной одному из типов памяти определяется:
  - местом объявления,
  - правилами умолчания,
  - параметрами директив.

СОЗДАНИЕ ПОТОКОВ

Формирование параллельной области

- Формат директивы **parallel** :

```
#pragma omp parallel [clause ...]
  structured_block
```

- Возможные параметры (**clauses**):

```
if (scalar_expression)
private (list)
firstprivate (list)
default (shared | none)
shared (list)
copyin (list)
reduction (operator: list)
num_threads(integer-expression)
```

Формирование параллельной области

- Директива parallel (основная директива OpenMP):
  - Когда основной поток выполнения достигает директивы parallel , создается команда (team) потоков.
  - Основной поток (master thread) входит в команду и имеет номер 0.
  - Код области parallel становится потоковой функцией и назначается потокам для параллельного выполнения.
  - В конце области автоматически обеспечивается синхронизация потоков.
  - Последующие вычисления продолжает выполнять только основной поток.

Пример использования директивы…

```
#include <omp.h>
void main() {
  int nthreads;
  // Создание параллельной области
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    // печать номера потока
    printf("Hello World from thread = %d\n", tid);
    // Печать количества потоков – только master
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } // Завершение параллельной области
}
```

Пример использования директивы

- Пример результатов выполнения программы для четырех потоков

Консоль отладки Microsoft Visual Studio

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 3
Hello World from thread = 2
```

Примечание: Порядок вывода на экран может меняться от запуска к запуску!!!

Установка количества потоков

- Способы задания (по убыванию старшинства)
  - Параметр директивы:
    `num_threads(N)`
  - Функция установки числа потоков:
    `omp_set_num_threads(N)`
  - Переменная окружения:
    `OMP_NUM_THREADS`
  - Число, равное количеству процессоров, которое "видит" операционная система.

Определение времени выполнения параллельной программы

```
double t1, t2, dt;
t1 = omp_get_wtime ();
…
t2 = omp_get_wtime ();
dt = t2 - t1;
```

Управление областью видимости

- Управление областью видимости обеспечивается при помощи параметров директив:
  - `shared`, `default`
  - `private`
  - `firstprivate`
  - `lastprivate`

- Параметры директив определяют, какие соотношения существуют между переменными последовательных и параллельных фрагментов выполняемой программы.

Параметр shared

- Параметр **`shared`** определяет список переменных, которые будут общими для всех потоков параллельной области.

```
#pragma omp parallel shared(list)
```

Примечание: правильность использования таких переменных должна обеспечиваться программистом.

Параметр private

- Параметр **`private`** определяет список переменных, которые будут локальными для каждого потока.

```
#pragma omp parallel private(list)
```

- Переменные создаются в момент формирования потоков параллельной области.
- Начальное значение переменных является неопределенным.

Пример использования директивы private

```
#include <omp.h>
void main() {
  int nthreads, tid;
  // Создание параллельной области
#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    // печать номера потока
    printf("Hello World from thread = %d\n", tid);
    // Печать количества потоков – только master
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } // Завершение параллельной области
}
```

Параметр firstprivate

- Параметр **`firstprivate`** позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных.

```
#pragma omp parallel firstprivate(list)
```

Параметр lastprivate

- Параметр **lastprivate** позволяет создать локальные переменные потоков, значения которых запоминаются в исходных переменных после завершения параллельной области (используются значения потока, выполнившего последнюю итерацию цикла или последнюю секцию).

```
#pragma omp parallel lastprivate(list)
```

БИБЛИОТЕКА ФУНКЦИЙ

Функции управления выполнением…

- Задать число потоков в параллельных областях

```
void omp_set_num_threads(int num_threads)
```

- Вернуть число потоков в параллельной области

```
int omp_get_num_threads(void)
```

- Вернуть максимальное число потоков, которое может быть создано в следующих параллельных областях без параметра **num_threads**

```
int omp_get_max_threads(void)
```

Функции управления выполнением…

- Вернуть номер потока в параллельной области

```
int omp_get_thread_num(void)
```

- Вернуть число процессоров, доступных приложению

```
int omp_get_num_procs(void)
```

- Возвращает true, если вызвана из параллельной области программы

```
int omp_in_parallel(void)
```

Вложенный параллелизм

- Включить/выключить вложенный параллелизм

```
int omp_set_nested(int)
```

- Вернуть, включен ли вложенный параллелизм

```
int omp_get_nested(void)
```

ПРИВЯЗКА ПОТОКОВ

Параметр proc_bind

- У директивы **parallel** в стандарте 4.0 был добавлен параметр proc_bind, определяющий способы «привязки» потоков к исполнительным устройствам
- Формат параметра **proc_bind**

```
#pragma omp parallel proc_bind(master | close |
  spread)
```

- master

Назначить все потоки в команде на то же устройство, на котором исполняется master-поток

Параметр proc_bind

- Формат параметра **proc_bind**

```
#pragma omp parallel proc_bind(master | close |
  spread)
```

- close

Распределить потоки по устройствам в порядке «деление по модулю»

- spread

Распределить потоки по устройствам «по блокам»

СПОСОБЫ РАСПРЕДЕЛЕНИЯ РАБОТЫ МЕЖДУ ПОТОКАМИ

Директивы распределения вычислений между потоками

- Существует 3 директивы для распределения вычислений в параллельной области:
  - **`for`** – распараллеливание циклов.
  - **`sections`** – распараллеливание раздельных фрагментов кода (функциональное распараллеливание).
  - **`single`** – директива для указания последовательного выполнения кода.
- Начало выполнения директив по умолчанию не синхронизируется.
- Завершение директив по умолчанию является синхронным.

Распараллеливание циклов

- Формат директивы **for**:

```
#pragma omp for [clause ...]
  for loop
```

- Основные параметры:
  - **private(list)**
  - **firstprivate(list)**
  - **lastprivate(list)**
  - **schedule(kind[, chunk_size])**
  - **reduction(operator: list)**
  - **nowait**

Пример использования директивы for

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
void main() {
  int i, n, chunk;
  float a[NMAX], b[NMAX], c[NMAX];
  for (i = 0; i < NMAX; i++)
    a[i] = b[i] = i * 1.0;
  n = NMAX; chunk = CHUNK;
#pragma omp parallel shared(a, b, c, n, chunk) private(i)
  {
#pragma omp for
    for (i = 0; i < n; i++)
      c[i] = a[i] + b[i];
  } // end of parallel section
}
```

Директива for. Параметр schedule

- **static** – итерации делятся на блоки по **chunk** итераций и статически разделяются между потоками.
  - Если параметр **chunk** не определен, итерации делятся между потоками равномерно и непрерывно.
- **dynamic** – распределение итерационных блоков осуществляется динамически (по умолчанию **chunk=1**).
- **guided** – размер итерационного блока уменьшается экспоненциально при каждом распределении.
  - **chunk** определяет минимальный размер блока (по умолчанию **chunk=1**).
- **runtime** – правило распределения определяется переменной **OMP_SCHEDULE** (при использовании **runtime** параметр **chunk** задаваться не должен).

Пример использования директивы for

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
void main() {
  int i, n, chunk;
  float a[NMAX], b[NMAX], c[NMAX];
  for (i = 0; i < NMAX; i++)
    a[i] = b[i] = i * 1.0;
  n = NMAX; chunk = CHUNK;
#pragma omp parallel shared(a, b, c, n, chunk) private(i)
  {
#pragma omp for schedule(static, chunk)
    for (i = 0; i < n; i++)
      c[i] = a[i] + b[i];
  } // end of parallel section
}
```

Объединение директив parallel и for/sections

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000

void main() {
  int i, n, chunk;
  float a[NMAX], b[NMAX], c[NMAX];
  for (i = 0; i < NMAX; i++)
    a[i] = b[i] = i * 1.0;
  n = NMAX; chunk = CHUNK;
#pragma omp parallel for shared(a, b, c, n) private(i) \
  schedule(static, chunk)
  for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

Распараллеливание циклов с редукцией

- Параметр **reduction** определяет список переменных, для которых выполняется операция редукции.

```
reduction (operator: list)
```

- Перед выполнением параллельной области для каждого потока создаются копии этих переменных.
- Потоки формируют значения в своих локальных переменных.
- При завершении параллельной области на всеми локальными значениями выполняется операция редукции, результаты которой запоминаются в исходных переменных.

Пример использования параметра reduction

```
#include <omp.h>
void main() {
  int i, n, chunk;
  float a[100], b[100], result;
  n = 100; chunk = 10;
  result = 0.0;
  for (i = 0; i < n; i++) {
    a[i] = i * 1.0; b[i] = i * 2.0;
  }
#pragma omp parallel for schedule(static, chunk) \
reduction(+: result)
  for (i = 0; i < n; i++)
    result = result + (a[i] * b[i]);
  printf("Final result= %f\n", result);
}
```

Правила записи параметра reduction

- Возможный формат записи выражения:
  - `x = x op expr`
  - `x = expr op x`
  - `x binop = expr`
  - `x++, ++x, x--, --x`
- **`x`** должна быть скалярной переменной.
- **`expr`** не должно ссылаться на **`x`**.
- **`op`** (operator) должна быть неперегруженной операцией вида:

  `+, -, *, &, ^, |, &&, ||`
- **`binop`** должна быть неперегруженной операцией вида:

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

Векторизация цикла…

- Директива **simd** – «просьба» компилятору векторизовать нижеследующий(-ие) цикл(-ы)

- Формат директивы **simd**

```
#pragma omp simd [clause ...]
  for_loops
```

Векторизация цикла

```
#pragma simd
#pragma omp parallel for private(d1, d2, erf1, erf2, invf)
for (i = 0; i < N; i++)
{
  invf = invsqrtf(sig2 * pT[i]);
  d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) * pT[i]) / invf;
  d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) * pT[i]) / invf;
  erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
  erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
  pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r * pT[i]) * erf2;
}
```

Директива threadprivate…

```
const int Size = 4;
int gmas[Size];
#pragma omp threadprivate(gmas)

void main()
{
  int i;
  for (i = 0; i < Size; i++)
    gmas[i] = Size - i;
  #pragma omp parallel
    if (omp_get_thread_num() == 1)
    {
      printf("\n");
      for (i = 0; i < Size; i++)
        printf("gmas[%d] = %d\n", i, gmas[i]);
      for (i = 0; i < Size; i++)
        gmas[i] = i;
    }
```

Директива threadprivate

```
 printf("\n");
  for (i = 0; i < Size; i++)
    printf("gmas[%d] = %d\n", i, gmas[i]);

  #pragma omp parallel
    if (omp_get_thread_num() == 1)
    {
      printf("\n");
      for (i = 0; i < Size; i++)
        printf("gmas[%d] = %d\n", i, gmas[i]);
    }

  #pragma omp parallel copyin(gmas)
    if (omp_get_thread_num() == 1)
    {
      printf("\n");
      for (i = 0; i < Size; i++)
        printf("gmas[%d] = %d\n", i, gmas[i]);
    }
}
```

СИНХРОНИЗАЦИЯ

Директива master

- Директива **`master`** определяет фрагмент кода, который должен быть выполнен только основным потоком
- Все остальные потоки пропускают данный фрагмент кода
- Завершение директивы не синхронизируется

```
#pragma omp master newline
  structured_block
```

Директива barrier

- Директива **barrier** определяет точку синхронизации, которую должны достигнуть все потоки для продолжения вычислений (директива должны быть вложена в блок)

- Формат директивы **barrier**

```
#pragma omp barrier newline
```

Директива single

- Директива **single** определяет фрагмент кода, который должен быть выполнен только одним потоком (любым)
- Один поток исполняет блок в **single**, остальные потоки приостанавливаются до завершения выполнения блока
- Формат директивы **single**

```
#pragma omp single [clause ...]
  structured_block
```

- Возможные параметры (**clauses**)

```
private(list)
firstprivate(list)
copyprivate(list)
nowait
```

Директива critical…

- Директива **`critical`** определяет фрагмент кода, который должен выполняться только одним потоком в каждый текущий момент времени (критическая секция)

- Формат директивы **`critical`**

```
#pragma omp critical [name] newline
  structured_block
```

Директива critical

```
#include <omp.h>
main()
{
  int x;
  x = 0;
#pragma omp parallel shared(x)
  {
#pragma omp critical
    x = x + 1;
  }
}
```

Директива atomic

- Директива **atomic** определяет переменную, операция с которой (чтение/запись) должна быть выполнена как неделимая
- Формат директивы **atomic**

```
#pragma omp atomic newline
  statement_expression
```

- Возможный формат записи выражения
  - x binop = expr , x++, ++x, x--, --x
  - x должна быть скалярной переменной
  - expr не должно ссылаться на x
  - binop должна быть неперегруженной операцией вида:
    +, -, *, /, &, ^, |, >>, <<

Директива atomic

```
#include <omp.h>
main()
{
  int x;
  x = 0;
#pragma omp parallel shared(x)
  {
#pragma omp atomic
    x = x + 1;
  }
}
```

Функции управления замками…

- В качестве замков используются переменные типа **`omp_lock_t`**.

- Инициализировать замок

```
void omp_init_lock(omp_lock_t *lock)
```

- Удалить замок

```
void omp_destroy_lock(omp_lock_t *lock)
```

Функции управления замками

- Захватить замок, если он свободен, иначе ждать освобождения

```
void omp_set_lock(omp_lock_t *lock)
```

- Освободить захваченный ранее замок

```
void omp_unset_lock(omp_lock_t *lock)
```

- Попробовать захватить замок. Если замок занят, возвращает false

```
int omp_test_lock(omp_lock_t *lock)
```

ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Функции управления выполнением…

- OMP_SCHEDULE – определяет способ распределения итераций в цикле, если в директиве for использована клауза schedule(runtime)
- OMP_NUM_THREADS – определяет число нитей для исполнения параллельных областей приложения
- OMP_NESTED – разрешает или запрещает вложенный параллелизм
- OMP_STACKSIZE – задать размер стека для потоков
- Компилятор с поддержкой OpenMP определяет макрос “_OPENMP”, который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы

ВОПРОСЫ?