ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В СТАНДАРТЕ OpenMP

Общая сумма разума на планете есть величина постоянная, несмотря на постоянный прирост населения.

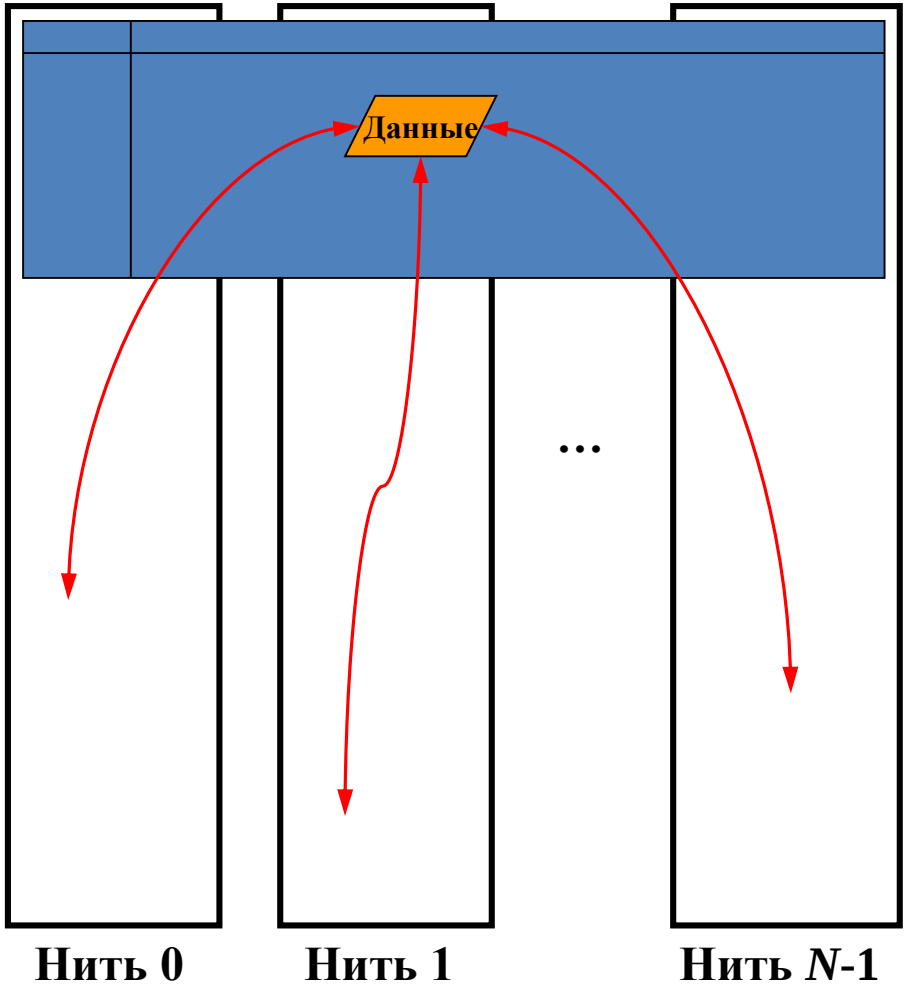
А. Блох

Суперкомпьютеры и их применение

Содержание

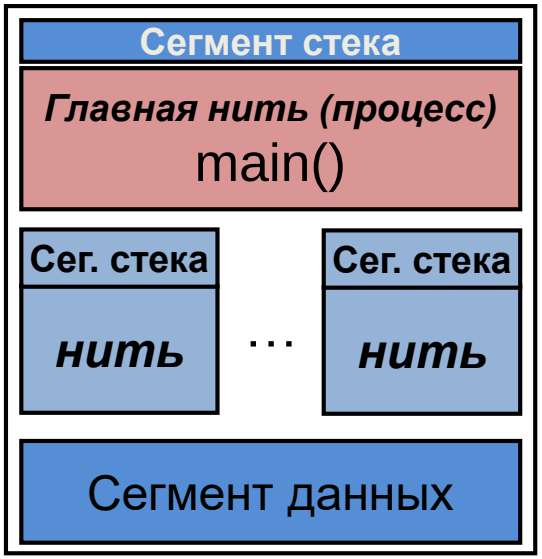
- Модель программирования в общей памяти
- Модель "пульсирующего" параллелизма FORK-JOIN
- Стандарт OpenMP
- Основные понятия и функции OpenMP

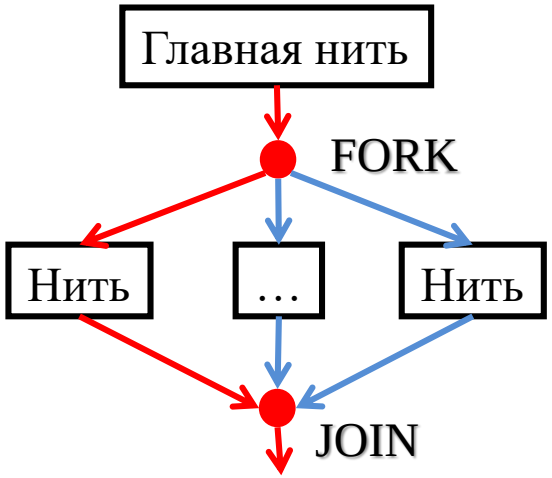
Программирование в общей памяти



- *Параллельное приложение* состоит из нескольких *нитей*, выполняющихся одновременно.
- Нити разделяют общую память.
- Обмены между нитями осуществляются посредством чтения/записи данных в общей памяти.
- Нити, как правило, выполняются на различных ядрах одного процессора.

Модель FORK-JOIN





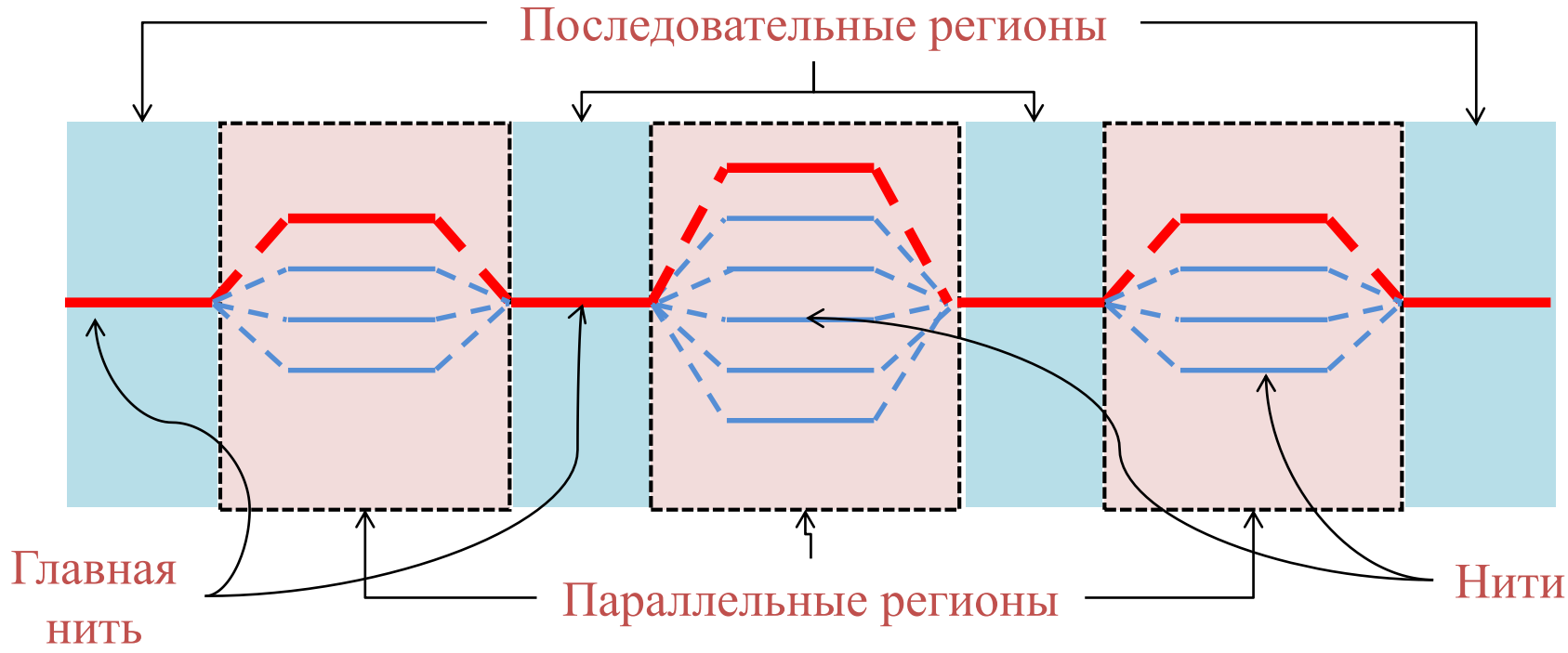
- Современные ОС поддерживают полновесные *процессы* (программы) и *легковесные процессы (нити)*.
  - Процесс приложения – *главная нить*.
  - Нить может запускать другие нити в рамках процесса. Каждая нить имеет собственный сегмент стека.
  - Все нити процесса разделяют сегмент данных процесса.

Стандарт OpenMP

- *OpenMP (Open Multi-Processing)* – стандарт, реализующий модели программирования в общей памяти и Fork-Join.
- Стандарт представляет собой набор директив компилятора и спецификаций подпрограмм для на языках C, C++ и FORTRAN.
- Стандарт реализуется разработчиками компиляторов для различных аппаратно-программных платформ (кластеры, персональные компьютеры, …, Windows, Unix/Linux, …).
- Разработкой стандарта занимается организация OpenMP Architecture Review Board (www.openmp.org).

OpenMP-программа

- Главная нить (программа) порождает семейство дочерних нитей (сколько необходимо). Порождение и завершение осуществляется с помощью *директив компилятора*.



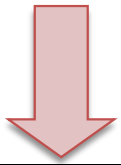
Простая OpenMP-программа

Последовательный код

```
void main()
{

printf("Hello!\n");
}
```

Результат

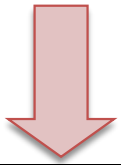


```
Hello!
```

Параллельный код

```
void main()
{
#pragma omp parallel
{
    printf("Hello!\n");
}
}
```

Результат (для 2-х нитей)



```
Hello!
Hello!
```

Преимущества OpenMP

- Поэтапное (инкрементное) распараллеливание
  - Можно распараллеливать последовательные программы поэтапно, не меняя их структуру.
- Единственность кода
  - Нет необходимости поддерживать последовательный и параллельный вариант программы, поскольку директивы игнорируются обычными компиляторами.
- Эффективность кода
  - Учет и использование возможностей систем с общей памятью.
- Мобильность кода
  - Поддержка языков C/C++, Fortran и ОС Windows, Unix/Linux.

Директивы OpenMP

- Директивы OpenMP – директивы C/C++ компилятора `#pragma`.
  - Для использования директив необходимо установить соответствующие параметры компилятора (обычно `-openmp`).
- Синтаксис директив OpenMP
  - `#pragma omp имя_директивы[параметры]`
- Примеры:
  - `#pragma omp parallel`
  - `#pragma omp for private(i, j) reduction(+: sum)`

Функции библиотеки OpenMP

- Назначение функций библиотеки:
  - контроль и просмотр параметров OpenMP-программы
    - `omp_get_thread_num()` возвращает номер текущей нити
  - явная синхронизация нитей на базе "замков"
    - `omp_set_lock()` устанавливает "замок"
- Подключение библиотеки
  - `#include "omp.h"`

Переменные окружения OpenMP

- Переменные окружения контролируют поведение приложения.
  - `OMP_NUM_THREADS` – количество нитей в параллельном регионе
  - `OMP_DYNAMIC` – разрешение или запрет динамического изменения количества нитей.
  - `OMP_NESTED` – разрешение или запрет вложенных параллельных регионов.
  - `OMP_SCHEDULE` – способ распределения итераций в цикле.
- Функции назначения параметров изменяют значения соответствующих переменных окружения.
- Макрос `_OPENMP` для условной компиляции отдельных участков исходного кода, характерных для параллельной версии программы.

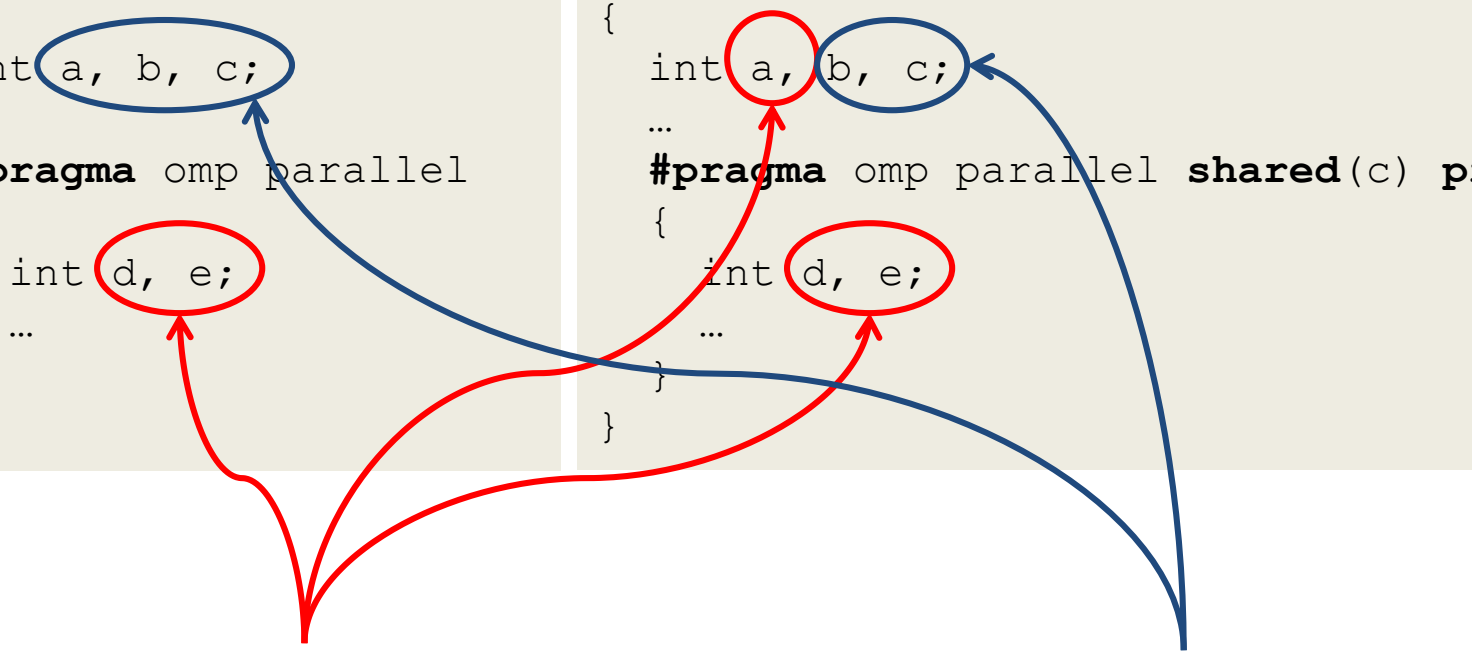
Область видимости переменных

- *Общая переменная (shared)* – глобальная по отношению к нити переменная; доступна для модификации всем нитям.
- *Частная переменная (private)* – локальная переменная нити; доступна для модификации только одной (создавшей ее) нити только на время выполнения этой нити.
- Видимость переменных по умолчанию:
  - переменные, определенные **вне** параллельной области – **общие**;
  - переменные, определенные **внутри** параллельной области – **частные**.
- Явное указание области видимости – параметры директив:
  - `#pragma omp parallel shared(buf)`
  - `#pragma omp for private(i, j)`

Частные и общие переменные


```
void main()
{
  int a, b, c;
  …
  #pragma omp parallel
  {
    int d, e;
    …
  }
}
```

```
void main()
{
  int a, b, c;
  …
  #pragma omp parallel shared(c) private(a)
  {
    int d, e;
    …
  }
}
```



Частные переменные

Общие переменные

Частные и общие переменные


```
void main()
{
  int rank;
 #pragma omp parallel
 {
    rank = omp_get_thread_num();
 }
 printf("%d\n", rank);
}
```

Одно (случайное) число из диапазона 0..OMP_NUM_THREADS-1

```
void main()
{
  int rank;
 #pragma omp parallel
 {
   rank = omp_get_thread_num();
   printf("%d\n", rank);
 }
}
```

OMP_NUM_THREADS случайных чисел (возможно, повторяющихся) из диапазона 0..OMP_NUM_THREADS-1

Частные и общие переменные


```
void main()
{
  int rank;
#pragma omp parallel shared (rank)
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

OMP_NUM_THREADS случайных чисел (возможно, повторяющихся) из диапазона 0..OMP_NUM_THREADS-1

```
void main()
{
  int rank;
 #pragma omp parallel private (rank)
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

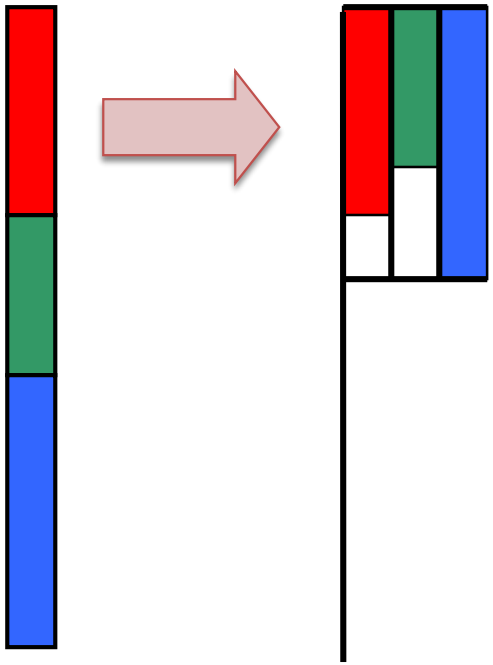
OMP_NUM_THREADS чисел из диапазона 0..OMP_NUM_THREADS-1 (без повторений, в случайном порядке)

Распределение вычислений

- Директивы распределения вычислений между нитями в параллельной области
  - `sections` – функциональное распараллеливание раздельных фрагментов кода
  - `single` и `master` – директивы для указания выполнения кода только одной нитью
  - `for` – распараллеливание циклов
- Начало выполнения директив по умолчанию не синхронизируется.
- Завершение директив по умолчанию является синхронным.

Директива sections


```
#pragma omp parallel sections
{
    #pragma omp section
    Job1();
    #pragma omp section
    Job2();
    #pragma omp section
    Job3();
}
```



- Явное определение блоков кода, которые могут исполняться параллельно.
  - каждый фрагмент выполняется однократно
  - разные фрагменты выполняются разными нитями
  - завершение директивы синхронизируется.

Директива `single`

- Определяет код, который выполняется только одной (первой пришедшей в данную точку) нитью.
  - Остальные нити пропускают соответствующий код и ожидают окончания его выполнения.
  - Если ожидание других нитей необязательно, может быть добавлен параметр `nowait`.

```
#pragma omp parallel
{
#pragma omp single
  printf("Start work #1.\n");
  Work1();
#pragma omp single
  printf("Stop work #1.\n");
#pragma omp single nowait
  printf("Stop work #1 and start work #2.\n");
  Work2();
}
```

Директива `master`

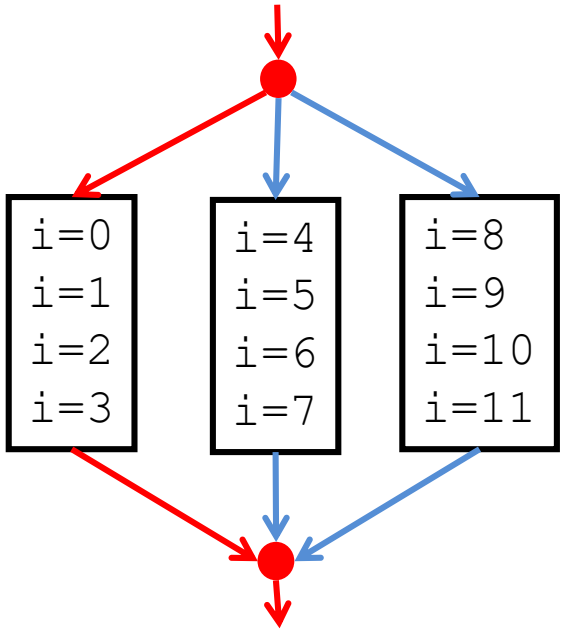
- Определяет код, который выполняется только одной главной нитью.
- Остальные нити пропускают соответствующий код, не ожидая окончания его выполнения.

```
#pragma omp parallel
{
#pragma omp master
  printf("Beginning work1.\n");
  work1();
#pragma omp master
  printf("Finishing work1.\n");
#pragma omp master
  printf("Finished work1 and beginning work2.\n");
  work2();
}
```

Распараллеливание циклов


```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++) {
    res[i] = big_calc();
  }
}
```

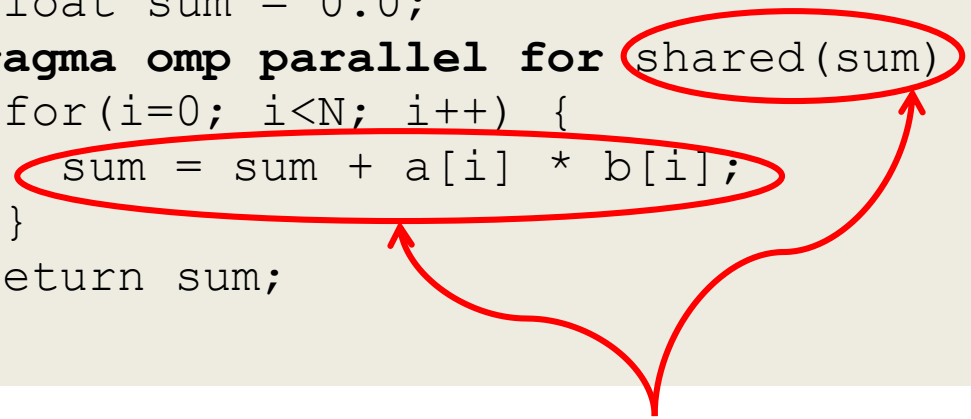
```
#pragma omp parallel for
for (i=0; i<N; i++) {
  res[i] = big_calc();
}
```



- Счетчик цикла по умолчанию является частной переменной.
- По умолчанию вычисления распределяются равномерно между нитями.

Распараллеливание циклов


```
float scalar_product(float a[], float b[], int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(i=0; i<N; i++) {
     sum = sum + a[i] * b[i];
   }
  return sum;
}
```



- Бесконтрольное изменение нитями общих данных приводит к логическим ошибкам.

Критическая секция в циклах


```
float scalar_product(float a[], float b[], int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(i=0; i<N; i++) {
#pragma omp critical
     sum = sum + a[i] * b[i];
   }
  return sum;
}
```

- В любой момент времени код критической секции может быть выполнен только одной нитью.

Редукция операций в циклах

- *Редукция* подразумевает определение для каждой нити частной переменной для вычисления "частичного" результата и автоматическое выполнение операции "слияния" частичных результатов.

```
float scalar_product(float a[], float b[], int N)
{
  float sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i++) {
    sum = sum + a[i] * b[i];
  }
  return sum;
}
```

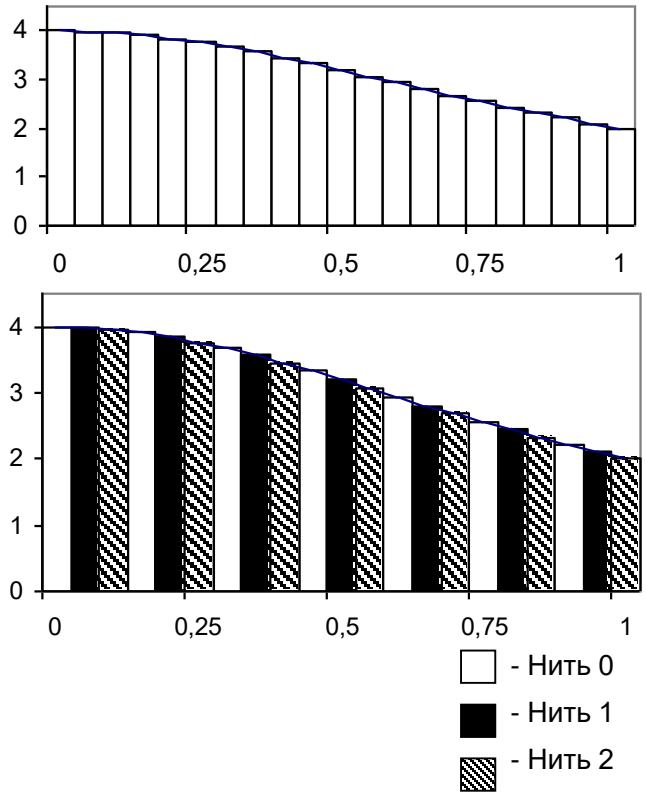
Операция	Нач. знач-е
+	0
*	1
-	0
^	0
&	~0
|	0
&&	1
||	0

Редукция операций в циклах


```
#include <stdio.h>
#include <time.h>
long long num_steps = 1000000000;
double step;
int main(int argc, char* argv[])
{
  clock_t start, stop;
  double x, pi, sum=0.0;
  int i;

  step = 1./(double)num_steps;
  start = clock();
#pragma omp parallel for private(x) reduction(+:sum)
  for (i=0; i<num_steps; i++) {
    x = (i + 0.5)*step;
    sum = sum + 4.0/(1.+ x*x);
  }
  pi = sum*step;
  stop = clock();
  printf("PI=%15.12f\n", pi);
  printf("Time=%f sec.\n",((double)(stop - start)/1000.0));
  return 0;
}
```

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



Директива `for`

- Формат директивы
  - `#pragma omp parallel for [clause ...]`
    `for (…)`
- Виды параметра `clause`
  - `private(список_переменных)`
  - `firstprivate(список_переменных)`
  - `lastprivate(список_переменных)`
  - `reduction(оператор: переменная)`
  - `ordered`
  - `nowait`
  - `schedule(вид_распределения[, размер])`

Параметр `firstprivate`

- Определяет частные переменные цикла `for` , которые в начале цикла принимают значения последовательной части программы.

```
myrank = omp_get_thread_num();
#pragma omp parallel for firstprivate(addendum)
for (i=0; i<N-1; i++) {
  a[i] = b[i] + b[i+1] + myrank;
  myrank = myrank + N % (i+1);
}
```

Параметр `lastprivate`

- Определяет частные переменные, которые по окончании цикла `for` принимают такие значения, как если бы цикл выполнялся последовательно.

```
#pragma omp parallel for lastprivate(i)
for (i=0; i<N-1; i++) {
  a[i] = b[i] + b[i+1];
}
// здесь i=N
```

Параметр `ordered`

- Определяет код в теле цикла `for`, выполняемый в точности в том порядке, в каком он выполнялся бы при последовательном исполнении цикла.

```
#pragma omp for ordered schedule(dynamic)
for (i=start; i<stop; i+=step)
  Process(i);

void Process(int k)
{
#pragma omp ordered
  printf(" %d", k);
}
```

Параметр `nowait`

- Позволяет избежать неявного барьера при завершении директивы `for`.

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
  for (i=0; i<m; i++)
    y[i] = sqrt(z[i]);
}
```

Распределение итераций цикла

- Распределение итераций в директиве `for` регулируется параметром `schedule (вид_распределения, [размер])`
  - `static` – итерации делятся на блоки по `размер` итераций и статически разделяются между потоками; если параметр `размер` не определен, итерации делятся между потоками равномерно и непрерывно
  - `dynamic` – распределение итерационных блоков осуществляется динамически (по умолчанию `размер=1`)
  - `guided` – размер итерационного блока уменьшается экспоненциально при каждом распределении; `размер` определяет минимальный размер блока (по умолчанию `размер=1`)
  - `runtime` – правило распределения определяется переменной `OMP_SCHEDULE` (при использовании `runtime` параметр `размер` задаваться не должен)

Пример


```
// Объем работы в итерациях предсказуем и примерно одинаков
#pragma omp parallel for schedule(static)
for(i=0; i<n; i++) {
  invariant_amount_of_work(i);
}
```

```
// Объем работы в итерациях может существенно различаться
// или непредсказуем
#pragma omp parallel for schedule(dynamic)
for(i=0; i<n; i++) {
  unpredictable_amount_of_work(i);
}
```

Пример


```
// Нити подходят к точке распределения итераций
// в разное время, объем работы в итерациях
// предсказуем и примерно одинаков
#pragma omp parallel
{
  #pragma omp sections nowait
  {
  ...
  }
  #pragma omp for schedule(guided)
  for(i=0; i<n; i++) {
    invariant_amount_of_work(i);
  }
}
```

Синхронизация вычислений

- Директивы явной синхронизации
  - `critical`
  - `barrier`
  - `atomic`
- Директива неявной синхронизации
  - `#pragma omp parallel`

Директива `critical`

- Определяет *критическую секцию* – участок кода, выполняемый одновременно не более чем одной нитью.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
#pragma omp critical (Xaxis_critical_section)
  x_next = Queue_Remove(x);
  Process(x_next);
#pragma omp critical (Yaxis_critical_section)
  y_next = Queue_Remove(y);
  Process(y_next);
}
```

Директива `atomic`

- Определяет *критическую секцию для одного оператора* вида
  - `x++` и `++x`
  - `x--` и `--x`
  - `x+=выражение`, `x-=выражение` и др.

```
extern float a[], *p = a, b;

// Предохранение от гонок данных
// при обновлении несколькими нитями
#pragma omp atomic
a[index[i]] += b;

// Предохранение от гонок данных
// при обновлении несколькими нитями
#pragma omp atomic
p[i] -= 1.0f;
```

Директива `barrier`

- Определяет *барьер* – точку в программе, которую должна достигнуть каждая нить, чтобы все нити продолжили вычисления.

```
#pragma omp parallel shared (A, TmpRes, FinalRes)
{
  DoSomeWork(A, TmpRes);
  printf("Processed A into TmpRes\n");
#pragma omp barrier
  DoSomeWork(TmpRes, FinalRes);
  printf("Processed B into C\n");
}
```

```
// Директива должна быть частью структурного блока
if (x!=0) {
  #pragma omp barrier
  ...
}
```

Директива barrier


```
int main()
{
sub1(2);
sub2(2);
sub3(2);
}
void sub1(int n)
{
int i;
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for (i=0; i<n; i++)
sub2(i);
}
}
void sub2(int k)
{
#pragma omp parallel shared(k)
sub3(k);
}
void sub3(int n)
{
work(n);
#pragma omp barrier
work(n);
}
```

Директивы и параметры

Параметр	Директива					
	parallel	for	sections	single	parallel for	parallel sections
if	✓				✓	✓
private	✓	✓	✓	✓	✓	✓
shared	✓	✓			✓	✓
default	✓				✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
reduction	✓	✓	✓		✓	✓
copyin	✓				✓	✓
schedule		✓			✓	
ordered		✓			✓	
nowait		✓	✓	✓		

Время работы


```
double start;
double end;


start = omp_get_wtime();
// Работа
end = omp_get_wtime();
printf("Work took %f sec. time.\n", end-start);
```

Количество нитей


```
// Неверно
np = omp_get_num_threads();
#pragma omp parallel for schedule(static)
for (i=0; i<np; i++)
  work(i);

// Верно

#pragma omp parallel private(i)
{
i = omp_get_thread_num();
work(i);
}
```

Функции синхронизации

- В качестве замков используются общие переменные типа `omp_lock_t`. Данные переменные должны использоваться только как параметры примитивов синхронизации.
- Инициализирует замок, связанный с переменной `lock`

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

- Удаляет замок, связанный с переменной lock

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

Функции синхронизации

- Заставляет вызвавшую нить дождаться освобождения замка, а затем захватывает его

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

- Освобождает замок, если он был захвачен нитью ранее

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

- Пробует захватить указанный замок. Если это невозможно, возвращает `false`

```
void omp_test_lock(omp_lock_t *lock)
```

Пример


```
#include <omp.h>
int main()
{
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);
  #pragma omp parallel shared(lck) private(id)
  {
  id = omp_get_thread_num();
  omp_set_lock(&lck);
  printf("My thread id is %d.\n", id);
 // only one thread at a time can execute this printf
  omp_unset_lock(&lck);
  while (! omp_test_lock(&lck)) {
   skip(id); /* we do not yet have the lock, so we must do something else */
  }
  work(id); /* we now have the lock and can do the work */
  omp_unset_lock(&lck);
  }
  omp_destroy_lock(&lck);
}
```

Заключение

- Модель программирования в общей памяти
- Модель "пульсирующего" параллелизма FORK-JOIN
- Стандарт OpenMP
- Основные понятия и функции OpenMP