
# 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

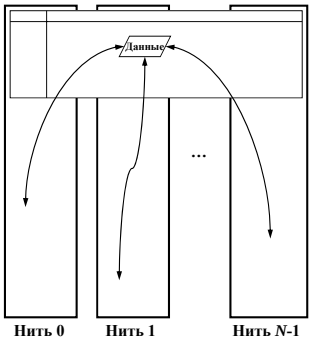
2

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



## Программирование в общей памяти

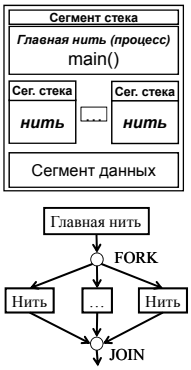
3




- *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

4







- 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

5

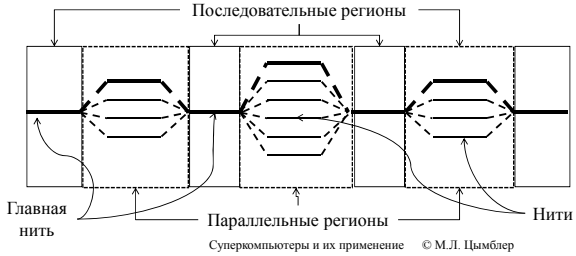
- *OpenMP* – стандарт, реализующий модели программирования в общей памяти и Fork-Join.
- Стандарт представляет собой набор директив компилятора и спецификаций подпрограмм для на языках C, C++ и FORTRAN.
- Стандарт реализуется разработчиками компиляторов для различных аппаратно-программных платформ (кластеры,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



# OpenMP-программа

6

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



# Простая OpenMP-программа

7

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

8

- 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

9

- 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

10

- 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

11

- 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.



## Управление областью видимости данных

12

- Параметр `shared(список)` определяет переменные которые будут *общими* для всех нитей.
  - Правильность использования общих переменных должна обеспечиваться программистом.
- Параметр `private(список)` определяет переменные, которые будут *локальными* для каждой нити.
  - Локальные переменные создаются в момент формирования потоков параллельной области, их начальное значение является неопределенным.

## Управление областью видимости данных

13

- Параметр `firstprivate(список)` создает локальные переменные нитей, которые перед использованием инициализируются значениями исходных переменных.
- Параметр `lastprivate(список)` создает локальные переменные нитей, значения которых запоминаются в исходных переменных после завершения параллельной области.
  - Используются значения из нити, выполнившей последнюю итерацию цикла или последнюю секцию.



## Распределение вычислений между нитями

14

- Директивы 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.



## Директива `for`

15

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

# Распределение итераций цикла

16

- Распределение итераций в директиве `for` регулируется параметром (`clause`) `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

17

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
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(dynamic,chunk) nowait
    for (i=0; i < n; i++)
      c[i] = a[i] + b[i];
  } // end of parallel section
}
```



# Директива `sections`

18

- Осуществляет распределение вычислений для раздельных фрагментов кода
  - фрагменты выделяются при помощи директивы `section`
  - каждый фрагмент выполняется однократно
  - разные фрагменты выполняются разными потоками
  - завершение директивы по умолчанию синхронизируется
  - директивы `section` должны использоваться только в статическом контексте

# Пример

19

```
#include <omp.h>
#define NMAX 1000
main () {
  int i, n;
  float a[NMAX], b[NMAX], c[NMAX];
  for (i=0; i < NMAX; i++)
    a[i] = b[i] = i * 1.0;
  n = NMAX;
  #pragma omp parallel shared(a,b,c,n) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i < n/2; i++)
          c[i] = a[i] + b[i];
      #pragma omp section
        for (i=n/2; i < n; i++)
          c[i] = a[i] + b[i];
    } // end of sections
  } // end of parallel section
}
```



# Директивы и параметры

20

| Clause | Directive | | | | | |
|---|---|---|---|---|---|---|
| | PARALLEL | DO/for | SECTIONS | SINGLE | PARALLEL DO/for | PARALLEL SECTIONS |
| IF | ● | | | | ● | ● |
| PRIVATE | ● | ● | ● | ● | ● | ● |
| SHARED | ● | ● | | | ● | ● |
| DEFAULT | ● | | | | ● | ● |
| FIRSTPRIVATE | ● | ● | ● | ● | ● | ● |
| LASTPRIVATE | | ● | ● | | ● | ● |
| REDUCTION | ● | ● | ● | | ● | ● |
| COPYIN | ● | | | | ● | ● |
| SCHEDULE | | ● | | | ● | |
| ORDERED | | ● | | | ● | |
| NOWAIT | | ● | ● | ● | | |



# Функции синхронизации

21

- 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

22

- 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)
```



## Заключение

23

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