Параллельное программирование в разделяемой памяти с использованием технологии OpenMP

Востокин Сергей Владимирович

План

- Общие сведения о технологии OpenMP
- Параллельные и последовательные области
- Модель данных
- Распределение работы
- Синхронизация
- Заключение

Литература: Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

Общие сведения о технологии OpenMP

Что такое OpenMP?

- OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран.
- Дает описание совокупности директив компилятора,
  библиотечных процедур и переменных окружения
- Предназначен для программирования многопоточных приложений на многопроцессорных системах с общей памятью

Разработка стандарта

C/C++ version 1.0 - (October 1998)

FORTRAN version 1.0 - (October 1997)

OpenMP 4.0 - (July 2013)

OpenMP 4.5 Complete Specifications - (November 2015)

Работа регулируется некоммерческой организацией, называемой OpenMP Architecture Review Board (ARB)

Сайт проекта: ***openmp.org***

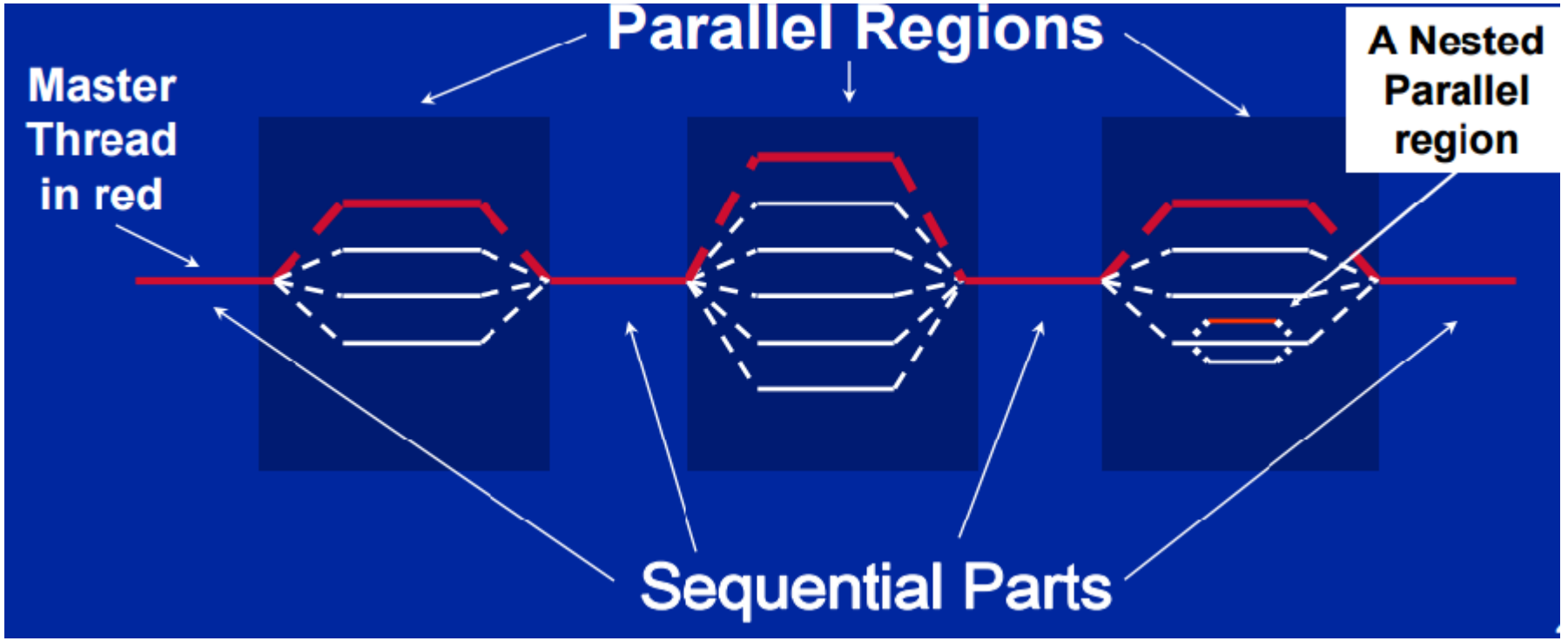
Компиляция программы

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
   printf("OpenMP is supported!\n");
#endif
}
```

Пример} OpenMP 3.0 определяет _OPENMP как 200805

Модель параллельной программы

В OpenMP предполагается SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код



Директивы и функции

Формат директивы на Си/Си++:

#pragma omp directive-name [опция[[,] опция]...]

Директива действует на ***оператор*** или ***ассоциированный блок***

Чтобы задействовать функции библиотеки OpenMP периода выполнения, в программу нужно включить заголовочный файл ***omp.h***

Замер времени

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    double start_time, end_time, tick;
    start_time = omp_get_wtime();
    end_time = omp_get_wtime();
    tick = omp_get_wtick();
    printf("Время на замер времени %lf\n", end_time-start_time);
    printf("Точность таймера %lf\n", tick);
}
```

Параллельные и последовательные области

Параллельная область

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Последовательная область 1\n");
#pragma omp parallel
    {
        printf("Параллельная область\n");
    }
    printf("Последовательная область 2\n");
}
```

Операция редукции

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int count = 0;
#pragma omp parallel reduction (+: count)
    {
        count++;
        printf("Текущее значение count: %d\n", count);
    }
    printf("Число нитей: %d\n", count);
}
```

Функция omp_set_num_threads() и опция num_threads

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
    {
        printf("Параллельная область 1\n");
    }
#pragma omp parallel
    {
        printf("Параллельная область 2\n");
    }
}
```

Функции omp_set_dynamic() и omp_get_dynamic()

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   printf("Значение OMP_DYNAMIC: %d\n", omp_get_dynamic());
   omp_set_dynamic(1);
   printf("Значение OMP_DYNAMIC: %d\n", omp_get_dynamic());
#pragma omp parallel num_threads(128)
   {
#pragma omp master
      {
         printf("Параллельная область, %d нитей\n",
                omp_get_num_threads());
      }
   }
}
```

Вложенные параллельные области (1/2)

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int n;
    omp_set_nested(1);
#pragma omp parallel private(n)
    {
        n=omp_get_thread_num();
#pragma omp parallel
        {
            printf("Часть 1, нить %d - %d\n", n,
                    omp_get_thread_num());
        }
    }
```

Вложенные параллельные области (2/2)

```
    omp_set_nested(0);
#pragma omp parallel private(n)
    {
        n=omp_get_thread_num();
#pragma omp parallel
        {
            printf("Часть 2, нить %d - %d\n", n,
                    omp_get_thread_num());
        }
    }
}
```

Функция *omp_in_parallel()*

```
#include <stdio.h>
#include <omp.h>
void mode(void){
   if(omp_in_parallel()) printf("Параллельная область\n");
   else printf("Последовательная область\n");
}
int main(int argc, char *argv[])
{
   mode();
#pragma omp parallel
   {
#pragma omp master
      {
         mode();
      }
   }
}
```

Директива *single* и опция *nowait*

```
#include <stdio.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
    {
        printf("Сообщение 1\n");
#pragma omp single nowait
        {
            printf("Одна нить\n");
        }
        printf("Сообщение 2\n");
    }
}
```

Опция *copyprivate*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int n;
#pragma omp parallel private(n)
    {
       n=omp_get_thread_num();
       printf("Значение n (начало): %d\n", n);
#pragma omp single copyprivate(n)
       {
          n=100;
       }
       printf("Значение n (конец): %d\n", n);
    }
}
```

Директива *master*

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int n;
#pragma omp parallel private(n)
    {
        n=1;
#pragma omp master
        {
            n=2;
        }
        printf("Первое значение n: %d\n", n);
#pragma omp barrier
#pragma omp master
        {
            n=3;
        }
        printf("Второе значение n: %d\n", n);
    }
}
```

Модель данных

Опция *private*

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
   int n=1;
   printf("n в последовательной области (начало): %d\n", n);
#pragma omp parallel private(n)
   {
      printf("Значение n на нити (на входе): %d\n", n);
      /* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (на выходе): %d\n", n);
   }
   printf("n в последовательной области (конец): %d\n", n);
}
```

Опция *shared*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i, m[10];
   printf("Массив m в начале:\n");
   /* Заполним массив m нулями и напечатаем его */
   for (i=0; i<10; i++){
      m[i]=0;
      printf("%d\n", m[i]);
   }
#pragma omp parallel shared(m)
   {
      /* Присвоим 1 элементу массива m, номер которого
      совпадает с номером текущий нити */
      m[omp_get_thread_num()]=1;
   }
   /* Ещё раз напечатаем массив */
   printf("Массив m в конце:\n");
   for (i=0; i<10; i++) printf("%d\n", m[i]);
}
```

Опция *firstprivate*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=1;
   printf("Значение n в начале: %d\n", n);
#pragma omp parallel firstprivate(n)
   {
      printf("Значение n на нити (на входе): %d\n", n);
      /* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (на выходе): %d\n", n);
   }
   printf("Значение n в конце: %d\n", n);
}
```

Директива *threadprivate*

```
#include <stdio.h>
#include <omp.h>
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{
   int num;
   n=1;
#pragma omp parallel private (num)
   {
      num=omp_get_thread_num();
      printf("Значение n на нити %d (на входе): %d\n", num, n);
      /* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити %d (на выходе): %d\n", num, n);
   }
   printf("Значение n (середина): %d\n", n);
#pragma omp parallel private (num)
   {
      num=omp_get_thread_num();
      printf("Значение n на нити %d (ещё раз): %d\n", num, n);
   }
}
```

Опция *copyin*

```
#include <stdio.h>
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[])
{
    n=1;
#pragma omp parallel copyin(n)
    {
        printf("Значение n: %d\n", n);
    }
}
```

Распределение работы

Функции *omp_get_num_threads()* и *omp_get_thread_num()*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int count, num;
#pragma omp parallel
    {
        count=omp_get_num_threads();
        num=omp_get_thread_num();
        if (num == 0) printf("Всего нитей: %d\n", count);
        else printf("Нить номер %d\n", num);
    }
}
```

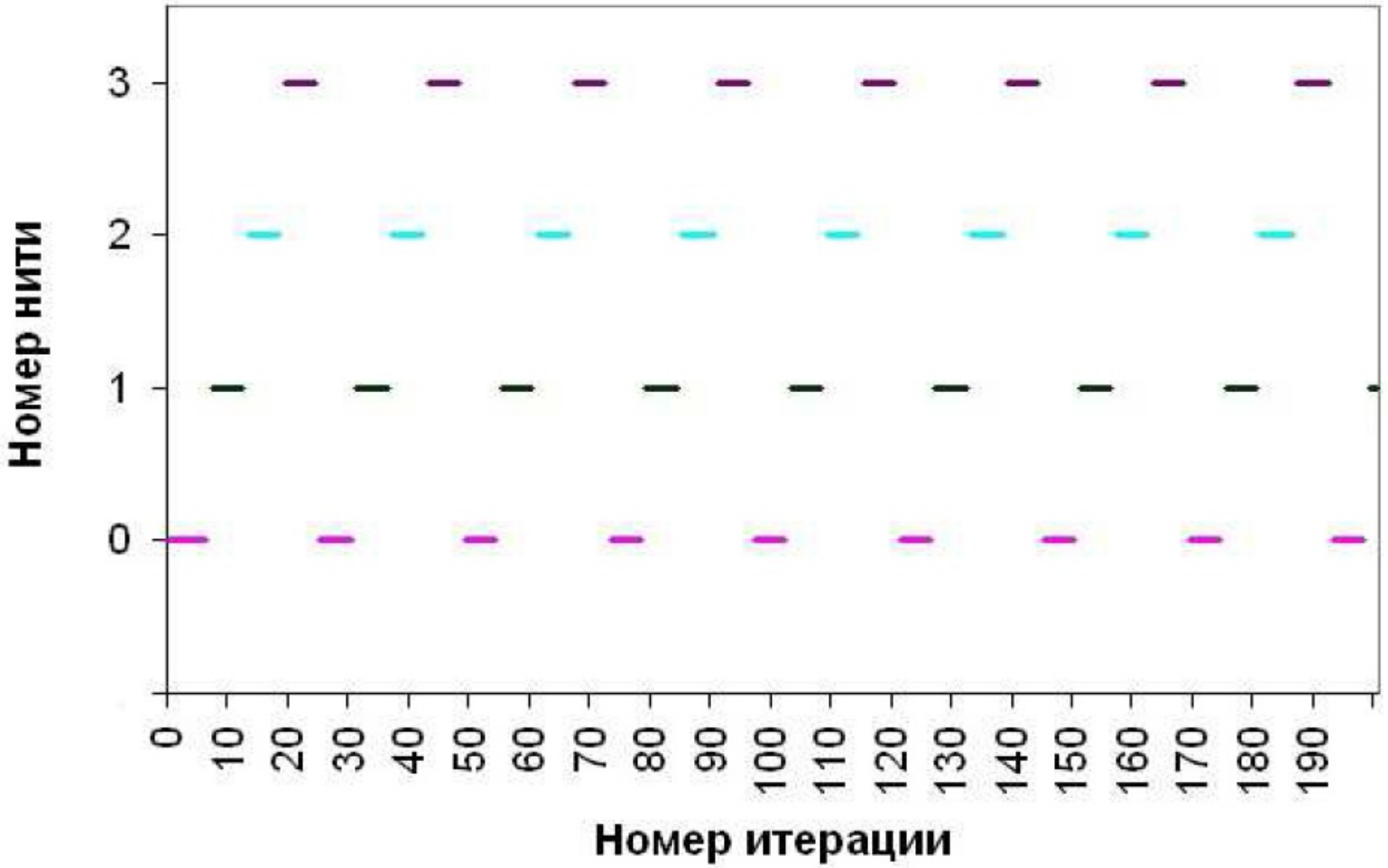
Директива *for*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int A[10], B[10], C[10], i, n;
/* Заполним исходные массивы */
   for (i=0; i<10; i++){ A[i]=i; B[i]=2*i; C[i]=0; }
#pragma omp parallel shared(A, B, C) private(i, n)
   {
      /* Получим номер текущей нити */
      n=omp_get_thread_num();
#pragma omp for
      for (i=0; i<10; i++)
      {
         C[i]=A[i]+B[i];
         printf("Нить %d сложила элементы с номером %d\n",
                n, i);
      }
   }
}
```

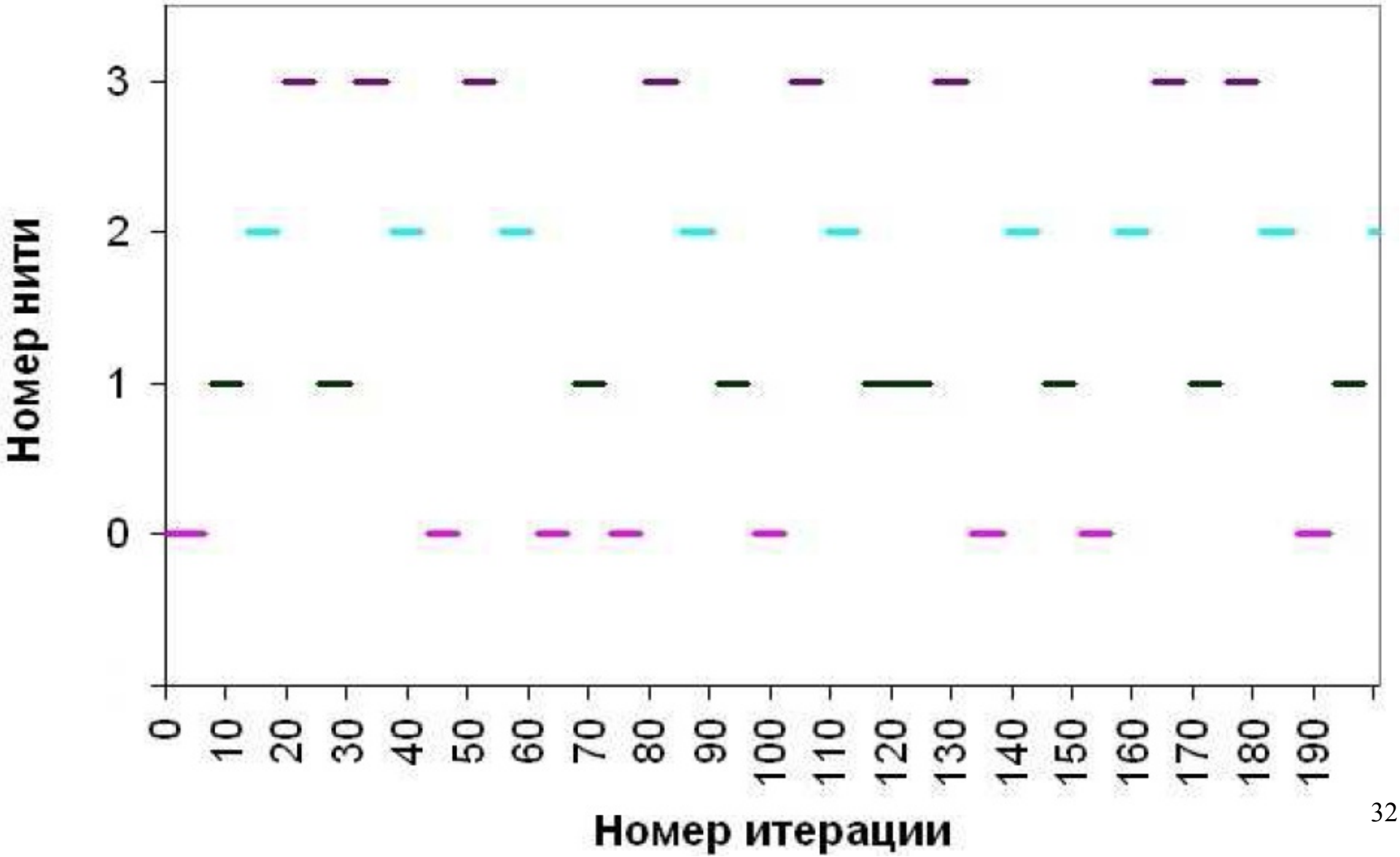
Опция *schedule*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i;
#pragma omp parallel private(i)
   {
#pragma omp for schedule (static, 6)
//#pragma omp for schedule (dynamic, 6)
//#pragma omp for schedule (guided, 6)
      for (i=0; i<200; i++)
      {
         printf("Нить %d выполнила итерацию %d\n",
                omp_get_thread_num(), i);
         sleep(1);
      }
   }
}
```

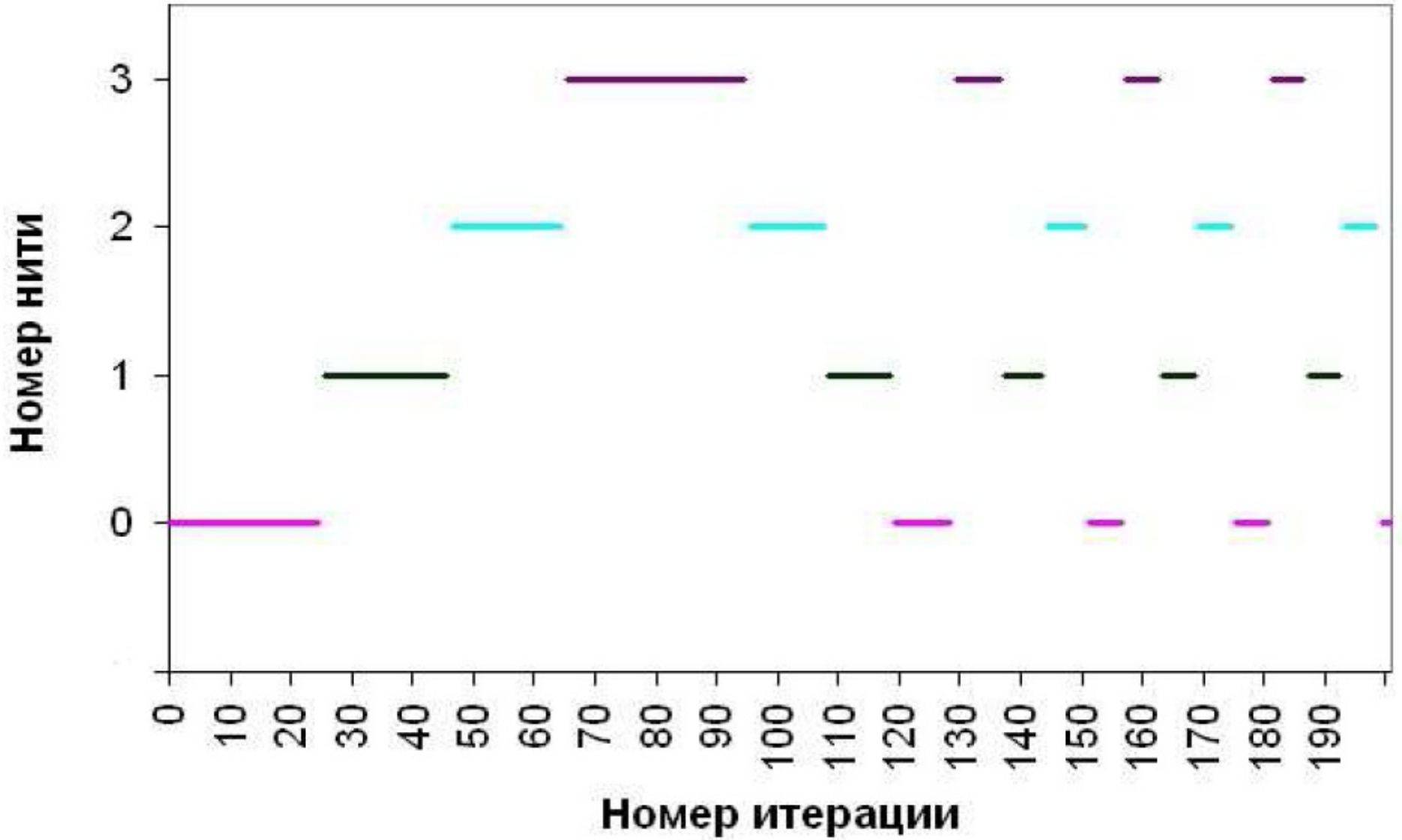
Распределение итераций по нитям для *(static, 6)*



Распределение итераций по нитям для *(dynamic, 6)*



Распределение итераций по нитям для *(guided, 6)*



Директива *sections*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int n;
#pragma omp parallel private(n)
    {
        n=omp_get_thread_num();
#pragma omp sections
        {
#pragma omp section
            {
                printf("Первая секция, процесс %d\n", n);
            }
#pragma omp section
            {
                printf("Вторая секция, процесс %d\n", n);
            }
#pragma omp section
            {
                printf("Третья секция, процесс %d\n", n);
            }
        }
        printf("Параллельная область, процесс %d\n", n);
    }
}
```

Опция *lastprivate*

```
    int n=0;
#pragma omp parallel
    {
#pragma omp sections lastprivate(n)
        {
#pragma omp section
            {
                n=1;
            }
#pragma omp section
            {
                n=2;
            }
#pragma omp section
            {
                n=3;
            }
        }
        printf("Значение n на нити %d: %d\n",
                omp_get_thread_num(), n);
```

Задачи

```
int fib ( int n )
{
int x,y;
 if ( n < 2 ) return n;
#pragma omp task shared (x)
 x = fib(n-1);
#pragma omp task shared(y)
 y = fib(n-2);
#pragma omp taskwait
 return x+y;
}
```

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
 for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
 process(e);
}
```

Синхронизация

Директива *barrier*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
    {
        printf("Сообщение 1\n");
        printf("Сообщение 2\n");
#pragma omp barrier
        printf("Сообщение 3\n");
    }
}
```

Директива *ordered* и опция *ordered*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int i, n;
#pragma omp parallel private (i, n)
   {
      n=omp_get_thread_num();
#pragma omp for ordered
      for (i=0; i<5; i++)
      {
         printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
         {
            printf("ordered: Нить %d, итерация %d\n", n, i);
         }
      }
   }
}
```

Директива *critical*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int n;
#pragma omp parallel
    {
#pragma omp critical
        {
            n=omp_get_thread_num();
            printf("Нить %d\n", n);
        }
    }
}
```

Директива *atomic*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int count = 0;
#pragma omp parallel
        {
#pragma omp atomic
            count++;
        }
        printf("Число нитей: %d\n", count);
}
```

Использование замков *lock()*

```
#include <stdio.h>
#include <omp.h>
omp_lock_t lock;
int main(int argc, char *argv[])
{
    int n;
    omp_init_lock(&lock);
#pragma omp parallel private (n)
    {
        n=omp_get_thread_num();
        omp_set_lock(&lock);
        printf("Начало закрытой секции, нить %d\n", n);
        sleep(5);
        printf("Конец закрытой секции, нить %d\n", n);
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
}
```

Функция *omp_test_lock()*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   omp_lock_t lock;
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      while (!omp_test_lock (&lock))
      {
         printf("Секция закрыта, нить %d\n", n);
         sleep(2);
      }
      printf("Начало закрытой секции, нить %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, нить %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Директива *flush*

```
#pragma omp parallel sections
 {
    #pragma omp section
    {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
    }
    #pragma omp section
    {
        #pragma omp flush (flag)
        while (flag == 0){
        #pragma omp flush (flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
    }
}
```

Заключение

Вычисление числа Пи

```
#include <stdio.h>
double f(double y) {return(4.0/(1.0+y*y));}
int main()
{
    double w, x, sum, pi;
    int i;
    int n = 1000000;
    w = 1.0/n;
    sum = 0.0;
#pragma omp parallel for private(x) shared(w)\
            reduction(+:sum)
    for(i=0; i < n; i++)
    {
        x = w*(i-0.5);
        sum = sum + f(x);
    }
    pi = w*sum;
    printf("pi = %f\n", pi);
}
```

Перемножение матриц

```
#include <stdio.h>
#include <omp.h>
#define N 4096
double a[N][N], b[N][N], c[N][N];
int main()
{
    int i, j, k;
    double t1, t2;
// инициализация матриц
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            a[i][j]=b[i][j]=i*j;
    t1=omp_get_wtime();
// основной вычислительный блок
#pragma omp parallel for shared(a, b, c) private(i, j, k)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            c[i][j] = 0.0;
            for(k=0; k<N; k++) c[i][j]+=a[i][k]*b[k][j];
        }
    }
    t2=omp_get_wtime();
    printf("Time=%lf\n", t2-t1);
}
```

Преимущества и недостатки (1/2)

- Переносимый многопоточный код
- Простота в использовании по сравнению с MPI
- Распределение данных и декомпозиция задаются автоматически директивами
- Масштабируемость сравнима с MPI на системах с общей памятью
- Инкрементное распараллеливание путем добавления директив в исходный последовательный код
- Одинаковый код для последовательного и параллельного приложения
- Исходный последовательный код не меняется при распараллеливании, это снижает вероятность ошибок
- Допускает мелкогранулярный параллелизм наряду с крупногранулярным
- Может использоваться с ускорителями, например с GPGPU и векторными операциями

Преимущества и недостатки (2/2)

- Риск привнесения сложно выявляемых ошибок синхронизации и гонок
- В настоящее время эффективен только на компьютерах с разделяемой памятью (хотя есть и распределённые реализации Cluster OpenMP)
- Требует поддержки компилятора
- Масштабируется на некоторых архитектурах памяти
- Нет поддержки примитива compare-and-swap
- Отсутствует надёжный механизм обработки ошибок
- Отсутствует механизм тонкого контроля распределения потоков по процессорам
- Риск написания кода с эффектом ложного разделения