

Интернет Университет Суперкомпьютерных технологий

Конструкции для синхронизации нитей

Учебный курс Параллельное программирование с OpenMP

Бахтин В.А., кандидат физ.-мат. наук,
заведующий сектором,
Институт прикладной математики им.
М.В.Келдыша РАН

Содержание

- ❑ Директива MASTER
- ❑ Директива CRITICAL
- ❑ Директива ATOMIC
- ❑ Семафоры
- ❑ Директива BARRIER
- ❑ Директива TASKWAIT
- ❑ Директива FLUSH
- ❑ Директива ORDERED

Директива master

```
#pragma omp master
```

структурный блок

*/*Структурный блок будет выполнен MASTER-нитью группы. По завершении выполнения структурного блока барьерная синхронизация нитей не выполняется*/*

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

```
void init(float *a, float *b ) {
```

```
    #pragma omp master
```

```
    scanf("%f %f", a, b);
```

```
    #pragma omp barrier
```

```
}
```

```
int main () {
```

```
    float x,y;
```

```
    #pragma omp parallel
```

```
    {
```

```
        init (&x,&y);
```

```
        parallel_work (x,y);
```

```
    }
```

```
}
```

Взаимное исключение критических интервалов

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

```
int i=0;  
#pragma omp parallel {  
    i++;  
}
```

Время	Thread0	Thread1
1	load i (i = 0)	
2	incr i (i = 1)	
3	->	load i (i = 0)
4		incr i (i = 1)
5		store i (i = 1)
6	store i (i = 1)	<-

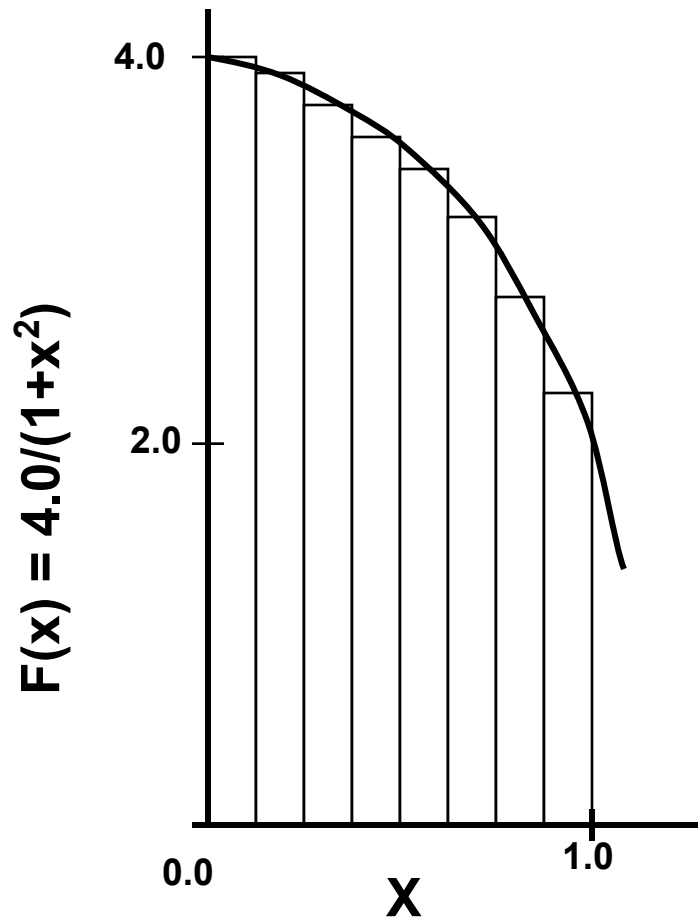
Результат зависит от порядка выполнения команд. Требуется взаимное исключение критических интервалов.

Взаимное исключение критических интервалов

Решение проблемы взаимного исключения должно удовлетворять требованиям:

- ❑ в любой момент времени только одна нить может находиться внутри критического интервала;
- ❑ если ни одна нить не находится в критическом интервале, то любая нить, желающая войти в критический интервал, должна получить разрешение без какой либо задержки;
- ❑ ни одна нить не должна бесконечно долго ждать разрешения на вход в критический интервал (если ни одна нить не будет находиться внутри критического интервала бесконечно);
- ❑ не должно существовать никаких предположений о скоростях процессоров.

Вычисление числа π



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Мы можем
аппроксимировать интеграл
как сумму прямоугольников:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Где каждый прямоугольник
имеет ширину Δx и высоту
 $F(x_i)$ в середине интервала

Вычисление числа π . Последовательная программа.

```
#include <stdio.h>
int main ()
{
    int n = 100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i++)
    {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP с использованием критической секции

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n = 100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
        double local_sum = 0.0;
        #pragma omp for
        for (i = 1; i <= n; i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
        }
        #pragma omp critical
            sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

#pragma omp critical [(name)]
структурный блок

Директива critical

```
int from_list(float *a, int type);  
void work(int i, float *a);
```

```
void example ()  
{  
#pragma omp parallel  
{  
    float *x;  
    int ix_next;  
    #pragma omp critical (list0)  
        ix_next = from_list(x,0);  
    work(ix_next, x);  
    #pragma omp critical (list1)  
        ix_next = from_list(x,1);  
    work(ix_next, x);  
}  
}
```

Директива atomic

#pragma omp atomic

expression-stmt

где expression-stmt:

- ☐ **x binop= expr**
- ☐ **x++**
- ☐ **++x**
- ☐ **x--**
- ☐ **--x**

Здесь x – скалярная переменная, expr – выражение со скалярными типами, в котором не присутствует переменная x.

где binop - не перегруженный оператор:

- ☐ **+**
- ☐ *****
- ☐ **-**
- ☐ **/**
- ☐ **&**
- ☐ **^**
- ☐ **|**
- ☐ **<<**
- ☐ **>>**

Вычисление числа π на OpenMP с использованием директивы `atomic`

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n = 100000, i;
    double pi, h, sum, x;
    h = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
        double local_sum = 0.0;
        #pragma omp for
        for (i = 1; i <= n; i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
        }
        #pragma omp atomic
        sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Семафоры

- ❑ Концепцию семафоров описал Дейкстра (Dijkstra) в 1965
- ❑ *Семафор* - неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций:
- ❑ P - функция запроса семафора
P(s): [if (s == 0) <заблокировать текущий процесс>; else s = s-1;]
- ❑ V - функция освобождения семафора
V(s): [if (s == 0) <разблокировать один из заблокированных процессов>; s = s+1;]

Семафоры в OpenMP

Состояния семафора:

- ❑ *uninitialized*
- ❑ *unlocked*
- ❑ *locked*

```
void omp_init_lock(omp_lock_t *lock); /* uninitialized to unlocked */  
void omp_destroy_lock(omp_lock_t *lock); /* unlocked to uninitialized */  
void omp_set_lock(omp_lock_t *lock); /*P(lock)*/  
void omp_unset_lock(omp_lock_t *lock); /*V(lock)*/  
int omp_test_lock(omp_lock_t *lock);
```

```
void omp_init_nest_lock(omp_nest_lock_t *lock);  
void omp_destroy_nest_lock(omp_nest_lock_t *lock);  
void omp_set_nest_lock(omp_nest_lock_t *lock);  
void omp_unset_nest_lock(omp_nest_lock_t *lock);  
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Вычисление числа π на OpenMP с использованием семафоров

```
#include <omp.h>
int main ()
{
    int n = 100000, i;
    double pi, h, sum, x;
    omp_lock_t lck;
    h = 1.0 / (double) n;
    sum = 0.0;
    omp_init_lock(&lck);
#pragma omp parallel default (none) private (i,x) shared (n,h,sum,lck)
    {
        double local_sum = 0.0;
#pragma omp for
        for (i = 1; i <= n; i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
        }
        omp_set_lock(&lck);
        sum += local_sum;
        omp_unset_lock(&lck);
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    omp_destroy_lock(&lck);
    return 0;
}
```

Использование семафоров

```
#include <stdio.h>
#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);
    return 0;
}
```

```
void skip(int i) {}
void work(int i) {}
```

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_lock_t lck; } pair;
void incr_a(pair *p, int a)
{
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_lock(&p->lck);
    p->b += b;
    omp_unset_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_lock(&p->lck);
}
```

```
void incorrect_example(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p,1,2);
        #pragma omp section
        incr_b(p,3);
    }
}
```

Deadlock!

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_nest_lock_t lck; } pair;
void incr_a(pair *p, int a)
{ /* Called only from incr_pair, no need to lock. */
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_nest_lock(&p->lck);
    /* Called both from incr_pair and elsewhere,
    so need a nestable lock. */
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
```

```
void correct_example(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p,1,2);
        #pragma omp section
        incr_b(p,3);
    }
}
```

Директива barrier

Точка в программе, достижимая всеми нитями группы, в которой выполнение программы приостанавливается до тех пор пока все нити группы не достигнут данной точки и все явные задачи, выполняемые группой нитей будут завершены.

#pragma omp barrier

По умолчанию барьерная синхронизация нитей выполняется:

- ❑ по завершению конструкции **parallel**
- ❑ при выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**) , если не указана клауза **nowait**.

#pragma omp parallel

```
{
    #pragma omp master
    {
        int i, size;
        scanf("%d",&size);
        for (i=0; i<size; i++) {
            #pragma omp task
            process(i);
        }
    }
    #pragma omp barrier
}
```

Директива barrier

```
void work(int i, int j) {}  
void wrong(int n)  
{  
    #pragma omp parallel default(shared)  
    {  
        int i;  
        #pragma omp for  
        for (i=0; i<n; i++) {  
            work(i, 0);  
            /* incorrect nesting of barrier region in a loop region */  
            #pragma omp barrier  
            work(i, 1);  
        }  
    }  
}
```

Директива barrier

```
void work(int i, int j) {}  
void wrong(int n)  
{  
    #pragma omp parallel default(shared)  
    {  
        int i;  
        #pragma omp critical  
        {  
            work(i, 0);  
            /* incorrect nesting of barrier region in a critical region */  
            #pragma omp barrier  
            work(i, 1);  
        }  
    }  
}
```

Директива barrier

```
void work(int i, int j) {}  
void wrong(int n)  
{  
    #pragma omp parallel default(shared)  
    {  
        int i;  
        #pragma omp single  
        {  
            work(i, 0);  
            /* incorrect nesting of barrier region in a single region */  
            #pragma omp barrier  
            work(i, 1);  
        }  
    }  
}
```

Директива taskwait

#pragma omp taskwait

```
int fibonacci(int n) {  
    int i, j;  
    if (n<2)  
        return n;  
    else {  
        #pragma omp task shared(i)  
        i=fibonacci (n-1);  
        #pragma omp task shared(j)  
        j=fibonacci (n-2);  
        #pragma omp taskwait  
        return i+j;  
    }  
}
```

```
int main () {  
    int res;  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            int n;  
            scanf("%d",&n);  
            #pragma omp task shared(res)  
            res = fibonacci(n);  
        }  
    }  
    printf ("Finonacci number = %d\n", res);  
}
```

Директива flush

#pragma omp flush [(список переменных)]

По умолчанию все переменные приводятся в консистентное состояние (**#pragma omp flush**):

- ☐ При барьерной синхронизации
- ☐ При входе и выходе из конструкций **parallel**, **critical** и **ordered**.
- ☐ При выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**) , если не указана клауза **nowait**.
- ☐ При вызове **omp_set_lock** и **omp_unset_lock**.
- ☐ При вызове **omp_test_lock**, **omp_set_nest_lock**, **omp_unset_nest_lock**
- ☐ и **omp_test_nest_lock**, если изменилось состояние семафора.

При входе и выходе из конструкции **atomic** выполняется **#pragma omp flush(x)**, где **x** – переменная, изменяемая в конструкции **atomic**.

Спасибо за внимание!

Вопросы?

Следующая тема

- ❑ Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы.

Контакты

□ **Бахтин В.А.**, кандидат физ.-мат. наук, заведующий сектором, Институт прикладной математики им. М.В. Келдыша РАН

bakhtin@keldysh.ru