

1. Приведите примеры параллельных алгоритмов вычисления суммы последовательности числовых значений.

```
#pragma omp parallel for schedule (static, 100000) reduction (+:sum)
#pragma omp parallel for schedule (dynamic, 100000) reduction (+:sum)
#pragma omp parallel for schedule (guided, 100000) reduction (+:sum)
#pragma omp parallel sections {
    #pragma omp section { sum1 = sum1 + 4.0 / (1.0 + x * x); }
    #pragma omp section { sum2 = sum2 + 4.0 / (1.0 + x * x); }
    #pragma omp section { sum3 = sum3 + 4.0 / (1.0 + x * x); }
    #pragma omp section { sum4 = sum4 + 4.0 / (1.0 + x * x); }
}
double pi = step * (sum1+sum2+sum3+sum4);
```

2. Каким образом происходит распределение работы между параллельными потоками?

Разделения работы между потоками

Балансировка загрузки параллельного исполнение цикла **for**

#pragma omp for дополнительные параметры:

- schedule** – планировщик распределения итераций цикла между потоками
 - schedule(static, chunk)** – статическое распределение
 - schedule(dynamic, chunk)** – динамическое распределение
 - schedule(guided, chunk)** – управляемое распределение
 - schedule(runtime)** – определяется OMP_SCHEDULE
- nowait** – отключение синхронизации в конце цикла
- ordered** – выполнение итераций в последовательном порядке

3. Охарактеризуйте особенности организации параллельной обработки с использованием секций.

4. Раскройте особенности балансировки нагрузки вычислений в параллельных секциях.

Балансировка загрузки

- ❑ Важные аспект производительности
- ❑ Для обычных операций (например, векторное сложение) балансировка редко нужна
- ❑ Для менее регулярных операций требуется балансировка
 - Примеры:
 - Транспонирование матриц
 - Умножение треугольных матриц
 - Поиск в списке

5. Охарактеризуйте планировщик распределения итераций цикла между потоками.

schedule(runtime) – планирование определяется через переменную окружения OMP_SCHEDULE

Решение относительно планирования не принимается, пока программа не загружена.
Тип **schedule** и размер блока может быть выбран на этапе запуска через переменную окружения OMP_SCHEDULE

6. Охарактеризуйте статическое распределение нагрузки.

Разделение работы между потоками

schedule(static, chunk) – статическое распределение

Итерации делятся на части, размер которых определен в **chunk**. Части статически заданны для нитей в группе в круговой циклической форме в порядке следования номеров нитей.

По умолчанию размер части: одна непрерывная часть для каждого потока. (пример для четырёх потоков)



7. Охарактеризуйте динамическое распределение нагрузки.

schedule(dynamic, chunk) – динамическое распределение

Итерации разбиваются на части, размера **chunk**. Как только каждый поток заканчивает часть, он динамически получает следующую часть.

По умолчанию размер части: 1.



8. Охарактеризуйте управляемое распределение нагрузки.

Разделение работы между потоками

schedule(guided, chunk) – управляемое распределение

Размер части уменьшается экспоненциально на каждом итерационном шаге. **chunk** определяется как наименьшая из возможных частей.

По умолчанию размер части: 1.



9. Что нужно сделать для выделения упорядоченного блока в распараллеленном цикле?

Динамически выделить память для двумерных массивов

10. Охарактеризуйте директивы синхронизации потоков **atomic**, **barrier**.

Синхронизация потоков

Директивы синхронизации потоков:

- **critical**
- **atomic**
- **barrier**
- **master**
- **single**
- **flush**
- **ordered**

```
Атомарная операция
int i, index[N], x[M];

#pragma omp parallel for \
    shared(index, x)
for (i=0; i<N; i++)
{
    #pragma omp atomic
    x[index[i]] += count(i);
}
```

Синхронизация потоков

Директивы синхронизации потоков:

- critical
- atomic
- **barrier**
- master
- single
- flush
- ordered

Барьер

```
int i;  
  
#pragma omp parallel for  
for (i=0; i<1000; i++)  
{  
    printf("%d ", i);  
    #pragma omp barrier  
}
```

Разделения работы между потоками barrier, ситуация для применения

Предположим мы выполняем следующий код:

```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];  
for (i=0; i < N; i++)  
    d[i] = a[i] + b[i];
```

Если циклы выполнять параллельно, то может быть неправильный ответ

- ❑ Нужна синхронизация по доступу к **a[i]**

Каждый поток ждет, пока все потоки достигнут определенную точку:
– #pragma omp barrier

11. Как происходит согласование значений переменных между потоками?

Синхронизация потоков

Директивы синхронизации потоков:

- critical
- atomic
- barrier
- master
- single
- **flush**
- ordered

Согласование значения переменных между потоками

```
int x = 0;  
#pragma omp parallel sections \  
shared(x)  
{  
    #pragma omp section  
    { x=1;  
      #pragma omp flush  
    }  
    #pragma omp section  
    while (!x);  
}
```

12. Поясните подходы к реализации параллельно-последовательного вычисления числа Pi.

```
//posled and paral realiz number pi  
void par_plus_posl(long N, long num_steps, float sum) {  
    long p4 = num_steps / N;  
    double step = 1.0 / (double)num_steps;  
#pragma omp parallel sections {  
#pragma omp section {  
    for (long i = 0; i < p4; i++) {  
        float x = (i + 0.5)*step;  
        sum = sum + 4.0 / (1.0 + x * x);  
    } }  
#pragma omp section {  
    for (long i = p4; i < num_steps; i++) {  
        float x = (i + 0.5)*step;  
        sum = sum + 4.0 / (1.0 + x * x);  
    } }  
    double pi = step * sum;  
    //printf("Pi = %f\n", pi);  
}
```

13. Приведите известные вам алгоритмы перемножения матриц.

1) через доп цикл прогона, где столбец*строку (не эффективно)

- 2) через доп цикл прогона, но вводится новая переменная, позже присваивается значение переменной определенной ячейки матрицы (эффективнее на 1-2 секунды)
- 3) через транспонирование матрицы (эффективнее на 4 сек)
- 4) алгоритм Штрассена
- 5) алгоритм Фокса
- 6) алгоритм Пана
- 7) алгоритм Бини
- 8) алгоритм Бини
- 9) алгоритм Щёнхаге
- 10) алгоритм Копперсмита-Винограда

14. Приведите рекомендации по разработке параллельных программ.

15. Охарактеризуйте функции блокировки (Lock) в OpenMP.

Функции блокировки (Lock) в OpenMP

Блокировки – более гибкий способ для управления критическими секциями:

- ❑ Возможно реализовать асинхронное поведение
- ❑ Используются специальные переменные C/C++ типы:
 omp_lock_t – простая блокировка (нельзя блокировать дважды)
 omp_nest_lock_t – вложенная блокировка (один поток может многократно блокировать переменную перед разблокированием)
- ❑ Можно управлять только через API
- ❑ Без инициализации переменных, поведение функций блокировок не определено

16. Поясните управление простой блокировкой (omp_lock_t).

Функции блокировки (Lock) в OpenMP

omp_lock_t (простая блокировка)

```
Инициализация блокировки  
void omp_init_lock(omp_lock_t *lock)  
  
Уничтожение блокировки  
void omp_destroy_lock(omp_lock_t *lock)  
  
Установка блокировки  
void omp_set_lock(omp_lock_t *lock)  
  
Снятие блокировки  
void omp_unset_lock(omp_lock_t *lock)  
  
Проверка на возможность установки и установка блокировки  
int omp_test_lock(omp_lock_t *lock)
```

17. Дайте определение понятию «задачи в OpenMP» и охарактеризуйте директиву task.

Понятие задачи. Директива task

Явные задачи (explicit tasks) задаются при помощи директивы:

```
#pragma omp task [опция [[,] опция ] ...]
```

структурный блок

где опция одна из :

- ☐ if (expression)
- ☐ untied
- ☐ shared (list)
- ☐ private (list)
- ☐ firstprivate (list)
- ☐ default(shared | none)

В результате выполнения директивы task создается новая задача, которая состоит из операторов структурного блока; все используемые в операторах переменные могут быть локализованы внутри задачи при помощи соответствующих опций. Созданная задача будет выполнена одним потоком из группы.

Понятие задачи. Директива task

Пример использования

```
typedef struct node node;
struct node {
    int data;
    node * next;
};
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            node * p = head;
            while (p) {
                #pragma omp task
                process(p); // p по умолчанию firstprivate
                p = p->next;
            }
        }
    }
}
```

Понятие задачи. Директива task

Пример использования с опцией if

```
double *item;
int main() {
    #pragma omp parallel shared (item)
    {
        #pragma omp single
        {
            int i, size;
            scanf("%d",&size);
            item = (double*)malloc(sizeof(double)*size);
            for (i=0; i<size; i++)
                #pragma omp task if (size > 10)
                process(item[i]);
        }
    }
}
```

Если накладные расходы на организацию задач превосходят время, необходимое для выполнения блока операторов этой задачи, то блок операторов будет немедленно выполнен потоком, выполнившим директиву task.

Понятие задачи. Директива task

Пример использования с опцией untied

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];

extern void process(double);

int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            int i;
            #pragma omp task untied
            {
                for (i=0; i<LARGE_NUMBER; i++)
                    #pragma omp task
                    process(item[i]);
            }
        }
    }
}
```

Опция untied - выполнение задачи после приостановки может быть продолжено любым потоком группы

18. Что означает понятие «взаимная блокировка» и когда она возникает?

Наиболее часто встречаемые проблемы

Взаимная блокировка директива critical

```
...
int A[N],B[N], sum;
#pragma omp parallel num_threads(10)
{
    int iam=omp_get_thread_num();
    if (iam ==0) {
        #pragma omp critical (update_a)
        #pragma omp critical (update_b)
        sum +=A[iam];
    } else {
        #pragma omp critical (update_b)
        #pragma omp critical (update_a)
        sum +=B[iam];
    }
}
...
```

Наиболее часто встречаемые проблемы

Взаимная блокировка Lock (вариант 1)

```
.....
#pragma omp parallel
{
    int iam=omp_get_thread_num();
    if (iam ==0) {
        omp_set_lock (&lcka);
        omp_set_lock (&lckb);
        x = x + 1;
        omp_unset_lock (&lckb);
        omp_unset_lock (&lcka);
    } else {
        omp_set_lock (&lckb);
        omp_set_lock (&lcka);
        x = x + 2;
        omp_unset_lock (&lcka);
        omp_unset_lock (&lckb);
    }
}
.....
```

© 2012, проект «686». All rights reserved. Все права защищены. Проект «686».

Наиболее часто встречаемые проблемы

Взаимная блокировка Lock (вариант 2)

```
.....
#pragma omp parallel
{
    int iam=omp_get_thread_num();
    if (iam ==0) {
        omp_set_lock (&lcka);
        while (x<0);
        /*цикл ожидания*/
        omp_unset_lock (&lcka);
    } else {
        omp_set_lock (&lcka);
        x++;
        omp_unset_lock (&lcka);
    }
}
}
```

Неинициализированные переменные, ошибка

```
#define N 100
#define Max(a,b) ((a)>(b)?(a):(b))

.....

float A[N], maxval, localmaxval;
maxval = localmaxval = 0.0;
#pragma omp parallel private (localmaxval)
{
    #pragma omp for
    for(int i=0; i<N;i++) {
        localmaxval = Max(A[i],localmaxval);
    }
    #pragma omp critical
    maxval = Max(localmaxval,maxval);
}
.....
```

Наиболее часто встречаемые проблемы

Неинициализированные переменные, исправление

```
#define N 100
#define Max(a,b) ((a)>(b)?(a):(b))

.....

float A[N], maxval, localmaxval;
maxval = localmaxval = 0.0;
#pragma omp parallel firstprivate (localmaxval)
{
    #pragma omp for
    for(int i=0; i<N;i++) {
        localmaxval = Max(A[i],localmaxval);
    }
    #pragma omp critical
    maxval = Max(localmaxval,maxval);
}
.....
```

Наиболее часто встречаемые проблемы

Неинициализированные переменные вариант 2

```
static int counter;
#pragma omp threadprivate(counter)

int main ()
{
    counter = 0;
    #pragma omp parallel
    {
        counter++;
    }
}
```

Наиболее часто встречаемые проблемы

Неинициализированные переменные вариант 2

```
static int counter;
#pragma omp threadprivate(counter)

int main ()
{
    counter = 0;
    #pragma omp parallel copyin (counter)
    {
        counter++;
    }
}
```

19. Охарактеризуйте проблему параллельного программирования – неинициализированные переменные.

20. Дайте определение понятию «зависимость данных и гонки в циклах».

Наиболее часто встречаемые проблемы

Зависимости данных и гонки в циклах

Ошибочный вариант реализации

```
#pragma omp parallel for
for (I=1; I<N; I++)
    a[I] = a[I-1] + heavy_func(I);
```

Корректная реализация

```
#pragma omp parallel for
for (I=1; I<N; I++)
    a[I] = heavy_func(I);

// последовательное выполнение
for (I=1; I<N; I++)
    a[I] += a[I-1];
```

21. Приведите параллельные алгоритмы умножения матрицы на вектор.

Умножение матрицы на вектор при разделении данных по строкам (paralell for)

Умножение матрицы на вектор при разделении данных по столбцам (omp_set_lock, paralell for)

```
#pragma omp parallel for reduction(+:IterGlobalSum)
```

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

```
#pragma omp critical
```

22. Приведите возможные виды распараллеливания алгоритмов перемножение матриц.

```
#pragma omp parallel for schedule (static, 10)
```

```
#pragma omp parallel for schedule (dynamic, 10)
```

```
#pragma omp parallel for schedule (guided, 10)
```

```
#pragma omp parallel for //if (size>120)
```

```
#pragma omp parallel for
```

```
#pragma omp parallel sections
```