

Удалённый доступ к кластеру

Для регистрации на удалённом кластере необходимо зайти по ssh на адрес `arch2018@ant.arpmath.spbu.ru` с паролем Chiez6Oong (Цифра - 6, остальные буквы). Для пользователей Windows можно использовать приложение Putty.

После регистрации нужно перезайти удалённо уже под вновь созданной учётной записью.

Работа с OpenMP

Компиляторы с поддержкой OpenMP можно посмотреть на официальном сайте – <http://www.openmp.org/resources/openmp-compilers/>.

Для работы с OpenMP можно использовать MS Visual Studio. После создания консольного приложения, нужно выбрать свойства проекта и включить поддержку OpenMP – properties/configuration properties/C.C++/language (рисунок 1).

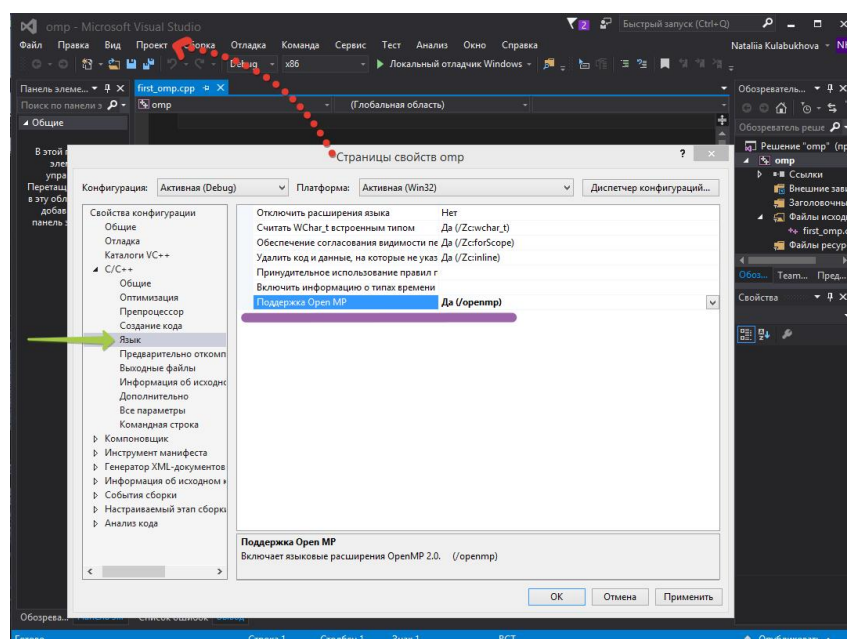


Рис. 1: Подключение интерфейса OpenMP в MS Visual Studio

Компиляция и запуск

Созданный файл с расширением `name.c` компилируется при помощи команды:

```
gcc -fopenmp name.c -o name.exe
```

Созданный файл с расширением name.cpp компилируется при помощи команды:

```
g++ -fopenmp name.c -o name.exe
```

Запуск программы и в том, и в другом случае осуществляется командой:

```
./name.exe
```

OpenMP директивы

Общий вид директив на языке C++ имеет вид:

```
# pragma omp конструкция [параметры]...
```

Особенности реализации директив

Количество потоков в программе определяется при помощи переменной окружения

```
export OMP_NUM_THREADS = 256
```

После выполнения этой команды, будет запущено 256 параллельных потоков. То же самое из программы можно задать при помощи функции `omp_set_num_threads()`. Для того чтобы узнать, какое количество нитей участвует в выполнении параллельного кода, существует функция `omp_get_num_threads()`. Так же количество параллельных потоков можно задать при описании директивы `#pragma omp parallel num_threads(n)`.

Каждый поток имеет свой номер `thread_number`. Определить номер текущего потока можно при помощи функции `omp_get_thread_num()`.

Пример использования функции:

```
# pragma omp parallel
{
    myid = omp_get_thread_num();
    if (myid == 0)
        do_something ()
    else
        do_something_esle (myid);
}
```

Существуют различные режимы выполнения параллельного блока программы:

динамический режим: задается из командной строки при помощи переменной окружения `OMP_DYNAMICS`. Если `true`, то режим включён, и `false`, если режим выключен.

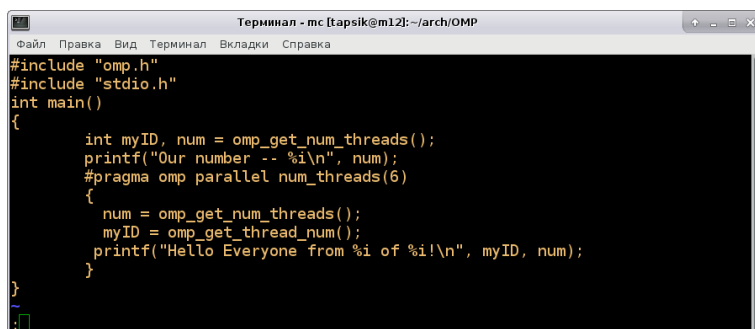
статический режим: количество потоков задается явно программистом. Сам режим устанавливается переменной окружения `OMP_STATIC`.

вложенный режим выполнения параллельных блоков: задается переменной окружения `OMP_NESTED`.

Все режимы можно задать и из программы при помощи соответствующих функций:

```
omp_set_dynamic(true|false);
omp_set_static(true|false);
omp_set_nested(true|false);
```

Пример простой программы:



```
Терминал - mc [tapsik@m12]: ~/arch/OMP
Файл  Правка  Вид  Терминал  Вкладки  Справка
#include "omp.h"
#include "stdio.h"
int main()
{
    int myID, num = omp_get_num_threads();
    printf("Our number -- %i\n", num);
    #pragma omp parallel num_threads(6)
    {
        num = omp_get_num_threads();
        myID = omp_get_thread_num();
        printf("Hello Everyone from %i of %i!\n", myID, num);
    }
}
```

Функции времени

Данные функции позволяют определить время выполнения конкретного блока кода, а не самой программы в целом.

```
double omp_get_wtime();
double omp_get_wtick();
```

Примеры программ с использованием различных типов задания переменных в параллельном блоке

Параметры основных директив OpenMP:

`# pragma omp parallel [parameter]`

`private (var 1, var 2, ...)` для каждой нити создается своя неинициализированная копия переменных `var1, var2` и т.д. Пример применения параметра `private` приведен на рисунке 2.

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

using namespace std;

int main()
{
    int n = 1;
    cout<<"N before parallel region = "<<n<<"\n";
    #pragma omp parallel private(n)
    {
        cout<<"N at first = "<<n<<"\n";
        n = omp_get_thread_num();
        cout<<"N after get_thread_num = "<<n<<"\n";
    }
    cout<<"N after parallel region = "<<n<<"\n";
}
```

Рис. 2: `private`

`shared (var 1, var 2, ...)` переменные `var1, var2` и т.д. общие для всех нитей. Пример применения параметра `shared` приведен на рисунке 3.

`firstprivate (var 1, var 2, ...)` для каждой нити создается своя локальная инициализированная копия переменной, значение переменных при выходе из параллельного блока не сохраняется. Пример применения параметра `firstprivate` приведен на рисунке 4.

`lastprivate (var 1, var 2, ...)` аналогично `firstprivate`, только глобальной переменной при выходе из параллельного блока присваивается значение переменной последней выполнившейся нити. Пример применения `lastprivate` показан на рисунке 5.

```

#include <omp.h>
#include <iostream>

using namespace std;

int main()
{
    int i, m[100];
    for (i = 0; i < 100; i++)
    {
        m[i] = 0;
        cout<<m[i]<<"\n";
    }
    #pragma omp parallel shared(m)
    {
        m[omp_get_thread_num()] = 1;
    }
    for (i = 0; i < 100; i++)
        cout<<"m["<<i<<" ] = "<<m[i]<<"\n";
}

```

Рис. 3: shared

`copyin (var 1, var 2, ...)` копирует значение переменной из главного потока в параллельные;

`reduction (operator : var 1, var 2, ...)` позволяет собрать в главном потоке результаты частичных сумм, разностей, умножений и т.д., применима к операциям `+`, `-`, `*`, `&`, `^`, `&&`, `max`, `min`. Является коллективной операцией над локальными переменными (операция редукции). Для переменной, используемой в данном случае, создаётся её локальная копия, затем полученные значения в потоках обрабатываются (например, суммируются) и запоминаются в исходной переменной. Использование общей переменной без создания локальных копий является неправильным, поскольку без обеспечения взаимного исключения при использовании общей переменной возникает ситуация гонки потоков и итоговый результат может быть ошибочным (рисунок 6).

`if (expression)` задает условие, при котором параллельный блок программы либо выполняется (если условие истинно), либо нет (если условие ложно);

`default (shared | none)` задает тип всех переменных в параллельном блоке по умолчанию `shared`, или он должен быть задан явно. При помощи этого параметра можно отменить действие правила по умолчанию (`default(none)`) или восстановить правило, вернув `default(shared)`.

Пример применения `threadprivate` на рисунке 7.

```

#include <omp.h>
#include <iostream>

using namespace std;

int main()
{
    int n = 1;
    cout<<"N before parallel region = "<<n<<"\n";
    #pragma omp parallel firstprivate(n)
    {
        cout<<"N at first = "<<n<<"\n";
        n = omp_get_thread_num();
        cout<<"N after get_thread_num = "<<n<<"\n";
    }
    cout<<"N after parallel region = "<<n<<"\n";
}

```

Рис. 4: firstprivate

Директива for

С помощью данной директивы можно осуществлять разделение итеративно-выполняемых действий в циклах для непосредственного указания, над какими данными выполняться соответствующие вычисления. Во многих случаях именно в циклах выполняется основная часть вычислительно-трудоемких вычислений.

```

# pragma omp parallel
{
# pragma omp for
for (i = 0; i < n; i++ )
{
...
}
}

```

или просто:

```
#pragma omp parallel for
```

После данной директивы итерации цикла распределяются между потоками, как результат, могут быть выполнены параллельно, когда между итерациями цикла нет информационной зависимости.

```

#include <iostream>
#include <omp.h>
#include <cmath>
using namespace std;
//-----
int main ()
{
    int n = 10, j;
    double x, a[n], b[n];
    for (int i=0; i<n; i++)
    { a[i] = i; b[i] = n-i; }
    cout<<"Mass a:"<<"\n";
    for (int i=0; i<n; i++)
    { cout<<a[i]<<"\n"; }
    cout<<"Mass b:"<<"\n";
    for (int i=0; i<n; i++)
    { cout<<b[i]<<"\n"; }
    #pragma omp parallel for lastprivate(x)
    for (j=0; j<n; j++)
    { x = a[j] + b[j]; b[j]= x; }
    cout<<"X value = "<<x<<"\n";
    cout<<"Mass b after lastprivate:"<<"\n";
    for (int i=0; i<n; i++)
    { cout<<b[i]<<"\n"; }
}

```

Рис. 5: lastprivate

Директива Schedule

Обеспечивает распределение работы между процессами. Общий вид:

```
#pragma omp schedule (type)
```

где type может быть:

static: блочно-циклическое распределение итераций цикла. Размер блока задается вручную явно или динамически машиной.

dynamic: динамическое распределение итераций с фиксированным размером блока. Сначала все нити получают свою порцию итераций, затем освободившаяся первой нить получает новую порцию и т.д.

guided: динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины указанной пользова-

```

int main ()
{
    int count=0;    int a[10], j, b=1;
    for(j = 0; j<10; j++)
        {a[j]=b; printf("%i \n", a[j]);}
    #pragma omp parallel shared (a, count) reduction(+: b)
    {
        {
            printf("At the beginning %i \n", b);
            count = omp_get_num_threads();
            int i;
            #pragma omp for
            for( i=0; i<5; i++)
                {
                    b = b + a[i];
                    printf("\n %i \n", b );
                }
        }
        printf( "\n %i \n", b);
        printf("Number of threads  %i \n", count);
    }
}

```

Рис. 6: Параметр reduction

телем явно или машиной автоматически, пропорционально количеству еще не распределённых итераций, деленному на количество нитей, выполняющих цикл.

auto: способ распределения итераций по нитям выбирается компилятором или системой. Параметры не задаются.

runtime: способ распределения итераций выбирается во время работы программы по значению переменной окружения "OMP_SCHEDULE"

Пример применения директивы с различными типами параметра type (рис. 8).

Директива sections

Данная директива задает участок кода, который будет выполняться одной нитью во время выполнения параллельного блока (рис. 9). Секции могут быть выполнены разными нитями или одной нитью. Порядок выполнения секций неопределён.


```

#include <omp.h>
#include <iostream>
using namespace std;
int n;
#pragma omp threadprivate (n)
int main ()
{
    int num;
    n = 1;
    #pragma omp parallel private (num)
    {
        num = omp_get_thread_num() ;
        cout<<"num - "<<num<<"\n"<<"n - "<<n<<"\n";
        n = omp_get_thread_num() ;
        cout<<"num - "<<num<<"\n"<<"n - "<<n<<"\n";
    }
    cout<<n<<"\n";
    #pragma omp parallel private (num)
    {
        num = omp_get_thread_num() ;
        cout<<"num - "<<num<<"\n"<<"n - "<<n<<"\n";
    }
}

```

Рис. 7: Объявление переменной как threadprivate()

Директива Single

Блок кода после директивы Single выполнится только в одном параллельном потоке (рис. 10). По-умолчанию, данная директива содержит барьер (директиву синхронизации), поэтому все потоки будут ждать выполнения данной секции, и только после этого продолжат выполнять свою работу дальше. Указав в директиве параметр nowait, мы говорим компилятору, что остальным потокам не нужно ждать выполнения секции Single и можно параллельно решать свои задачи. Применение данного параметра оправданно только в том случае, когда точно известно, что внесённые секцией Single изменения никак не повлияют на общие данные, используемые другими потоками.

Пример с использованием директивы Single и параметра copyprivate (рис. 11). Так называемая широковещательная рассылка, позволяющая передать изменённое в секции Single значение переменной, указанной как copyprivate, локальным переменным остальных потоков.

Синхронизация

В OpenMP существует шесть типов синхронизации:

```

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

//-----
int main ()
{
    int i;
    #pragma omp parallel private(i)
    {
        #pragma omp for schedule (static)
        // #pragma omp for schedule (static, 1)
        // #pragma omp for schedule (static, 5)
        // #pragma omp for schedule (dynamic)
        // #pragma omp for schedule (dynamic, 1)
        // #pragma omp for schedule (dynamic, 3)
        // #pragma omp for schedule (guided)
        // #pragma omp for schedule (auto)
        // #pragma omp for schedule (runtime)
        for (i=0; i<10; i++)
        {
            printf("Thread %d runs i = %i \n", omp_get_thread_num(), i);
        }
    }
}

```

Рис. 8: schedule

```

#include <omp.h>
#include <stdio.h>
//-----
int main ()
{
    int n;
    #pragma omp parallel private(n)
    {
        n = omp_get_thread_num();
        printf("Before pragma sections %i \n", n);
        #pragma omp sections
        {
            #pragma omp section
            {printf("First section runs on thread number - %i \n", n);}
            #pragma omp section
            {printf("Second section runs on thread number - %i \n", n);}
            #pragma omp section
            {printf("Third section runs on thread number - %i \n", n);}
        }
        printf("Parallel regeon, thread number - %i \n", n);
    }
}

```

Рис. 9: sections

```

#include <omp.h>
#include <stdio.h>
//-----
int main ()
{
    #pragma omp parallel num_threads(4)
    {
        printf("Message before Single \n");
        #pragma omp single //nowait
        { printf("Running on one thread \n");}
        printf("Message after Single \n");
    }
}

```

Рис. 10: single

```

#include <omp.h>
#include <stdio.h>
//-----
int main ()
{
    int n;
    #pragma omp parallel private(n)
    {
        n = omp_get_thread_num();
        printf("n is %i \n", n);
        #pragma omp single copyprivate(n)
        {n = 100;}
        printf("n after copyprivate - %i \n", n);
    }
}

```

Рис. 11: copyprivate

critical: используется для описания структурных блоков, выполняющихся только в одном потоке из всего набора параллельных потоков (рис. 12).

atomic: определяет переменную в левой части оператора присваивания, которая должна корректно обновляться несколькими нитями (альтернатива операции reduction)(рис. 13).

barrier: устанавливает режим ожидания завершения работы всех запущенных в программе параллельных потоков (рис. 14).

master: используется для определения структурного блока, который будет выполняться исключительно в главном потоке (нулевом потоке)

ordered: используется для определения потоков, которые выполняются в порядке, соответствующем последовательной версии программы (рис. 15).

flush: используется для обновления значений локальных переменных, перечисленных в качестве аргументов этой программы, в оперативной памяти.

Примеры перечисленных типов приведены ниже.

```
#include <omp.h>
#include <stdio.h>
//-----
int main ()
{
    int n;
    #pragma omp parallel private(n)
    {
        n = omp_get_thread_num();
        printf("Before critical %i \n", n);
        #pragma omp critical
        {
            //n = omp_get_thread_num();
            printf("Thread %i is working \n", n);
        }
        printf("After critical %i \n", n);
    }
}
```

Рис. 12: critical

Механизм замков (Locks)

Замок представляет собой целочисленную переменную, использующуюся для хранения адреса. Замок может находиться в одном из трех состояний:

- неинициализированный;
- разблокированный;
- заблокированный.

Так же замки бывают:

- простые;
- множественные.

```

int main ()
{
    int count=0, j; int a[10], b=1;
    for (j = 0; j<10; j++)
        {a[j]=b; printf("%i, ",a[j]);}
    printf("\n");
    #pragma omp parallel shared (a, b, count)
    {
        {
            int i;
            count = omp_get_num_threads();
            #pragma omp for
            for ( i=0;i<10;i++)
                {
                    #pragma omp atomic
                    b = b + a[i];
                    printf("b = %i \n", b);
                }
        }
    }
    printf("\n %i \n", b);
    printf("Number of threads = %i \n", count);
}

```

Рис. 13: atomic

```

int main ()
{
    #pragma omp parallel
    {
        printf("Before barrier\n");
        printf("Waiting for every thread %i \n", omp_get_thread_num());
        #pragma omp barrier
        printf("after the barrier\n");
    }
}

```

Рис. 14: barrier

В первом случае, замок может быть либо разблокирован, либо заблокирован одной нитью. С случае множественного замка вводится понятие коэффициента захваченности. Как только нить захватывает замок, коэффициент увеличивается на единицу, в случае разблокирования — коэффициент уменьшается так же на единицу. При этом захват могут производить несколько нитей. Если замок свободен (то есть ни одна нить не захватила этот замок), коэффициент захваченности множественного замка равен нулю.

Для инициализации замка используют следующие функции:

```

omp_init_lock();
omp_init_nest_lock();

```

```

int main ()
{
    int n,i;
    #pragma omp parallel private(n,i)
    {
        n = omp_get_thread_num();
        #pragma omp for ordered
        for (i=0;i<5;i++)
        {
            printf("Thread number %i runs %i \n",n, i);
            #pragma omp ordered
            {
                printf("Thread number %i runs ordered iteration %i \n", n, i);
            }
        }
    }
}

```

Рис. 15: ordered

Вторая функция — для инициализации множественного замка.

Обратные функции — это функции возвращающие замок в неинициализированное состояние.

```

omp_destroy_lock();
omp_destroy_nest_lock();

```

Для захвата замка и освобождения замка используются соответствующие функции:

```

omp_set_lock();
omp_set_nest_lock();

omp_unset_lock();
omp_unset_nest_lock();

```

Для неблокирующей попытки захвата замка применяется функция

```

omp_test_lock();
omp_test_nest_lock();

```

Если нити удалось захватить замок, то в случае простого замка будет возвращено значение 1, в случае множественного - коэффициент захваченности. Если попытка не удалась, будет возвращен 0 в обоих случаях.

Пример простого замка показан на рисунке 16. На рисунке 17 показано применение функции `omp_test_lock()`.

```

omp_lock_t lock;
//-----
int main ()
{
    printf("\n");
    int n,i;
    double sum,a[1000],b[1000];
    for (i=0;i<1000;i++)
    { a[i] = i+0.4*i; b[i] = 0.87*i;}
    omp_init_lock(&lock);
    #pragma omp parallel private(n,i)
    {
        n = omp_get_thread_num();
        printf("Before lock %i \n", n);
        omp_set_lock(&lock);
        printf("Beginning of close section, thread - %i \n", n);
        for (i=0;i<1000;i++)
            sum = sum + a[i]*b[i];
        printf("Ending of close section, thread - %i \n", n);
        omp_unset_lock(&lock);
        printf("After lock %i \n", n);
    }
    omp_destroy_lock;
}

```

Рис. 16: Locks

Дополнительные переменные среды и функции

В OpenMP можно задавать максимально допустимое количество вложенных параллельных областей как с помощью переменных окружения, так и соответствующими функциями:

```

OMP_MAX_ACTIVE_LEVEL
omp_set_max_active_level();
omp_get_max_active_level();

```

Просто количество вложенных параллельных областей в данном месте кода определяется функцией `omp_get_active_level()`.

`omp_get_level()` — выдает количество вложенных параллельных областей в данном месте кода.

Можно так же узнать, номер нити, породившей нити на следующем уровне вложенности:

```
omp_get_ancestor_thread_num(level);
```

С помощью `omp_get_team_size(level)` получим количество нитей, порожденных одной родительской нитью.


```

omp_lock_t lock;
//-----
int main ()
{
    int n,i;
    double sum,a[1000],b[1000];
    for (i=0;i<1000;i++)
    { a[i] = i+0.4*i; b[i] = 0.87*i; }
    omp_init_lock(&lock);
    #pragma omp parallel private(n,i) num_threads(3)
    {
        n = omp_get_thread_num();
        while (!omp_test_lock(&lock))
        { printf("Section is closed, thread - %i \n ", n); }
        printf("Beginning of close section, thread - %i \n", n);
        for (i=0;i<1000;i++)
            sum = sum + a[i]*b[i];
        printf("Ending of close section, thread - %i \n", n);
        omp_unset_lock(&lock);
    }
    omp_destroy_lock;
}

```

Рис. 17: Test Locks

Существуют и другие вспомогательные функции, которые можно найти в справочнике по OpenMP.