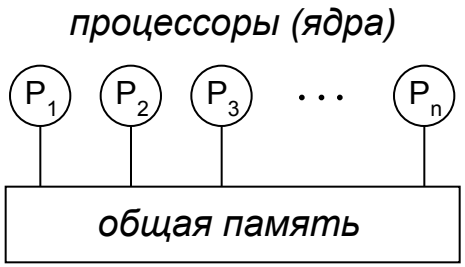
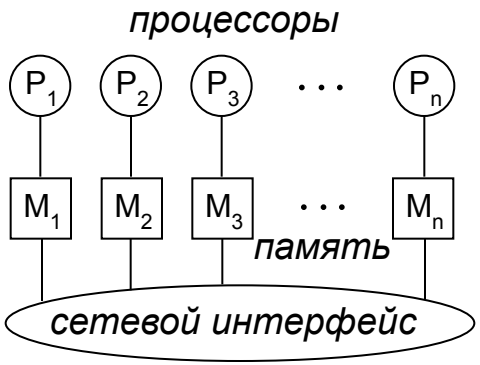
Технологии параллельного программирования

Развитые средства и технологии для разработки параллельных программ

- Автоматическое распараллеливание
- Библиотеки нитей (threads)
    - Win32/64 API
    - POSIX (Portable Operating System Interface for Unix)
- OpenMP (Open Multi-Processing)
- Библиотеки передачи сообщений
    - MPI (Message Passing Interface)
    - PVM (Parallel Virtual Machine)





Автоматическое распараллеливание

Автоматическое (автоматизированное) распараллеливание (*automatic parallelization*) предполагает выполнение оптимизации программного кода компилятором путем преобразования в форму, наиболее эффективно работающую на данной многопроцессорной системе.

Компиляторы с поддержкой автоматического распараллеливания:
Intel (OpenMP), GNU GCC, Polaris, CAPO, WPP, SUIF, VAST/Parallel, OSCAR, ParaWise

```
parallel_auto.f
parallel_auto.f(8): (col. 5) remark: LOOP WAS AUTO-PARALLELIZED.
parallel_auto.f(8): (col. 5) remark: LOOP WAS VECTORIZED.
parallel_auto.f(8): (col. 5) remark: LOOP WAS VECTORIZED.
```

В общем случае, автоматическое распараллеливание затруднено:

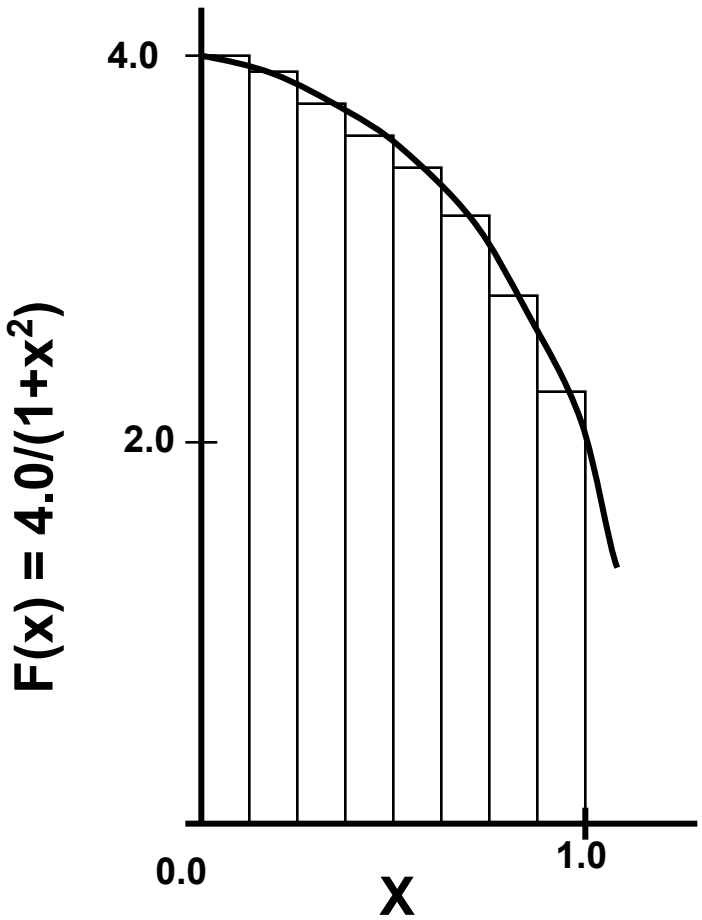
- косвенная индексация (A[B[i]])
- указатели (ассоциация по памяти)
- сложный межпроцедурный анализ
- циклы с зависимостью по данным, как правило не распараллеливаются

Библиотеки нитей. Пример программы

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello world from thread %ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t < NUM_THREADS; t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

pthread_exit(NULL);
}
```

```
% gcc -pthread -o hello_pthreads
hello_pthreads.c

% ./hello_pthreads
In main: creating thread 0
In main: creating thread 1
Hello World from thread 0!
In main: creating thread 2
Hello World from thread 1!
In main: creating thread 3
Hello World from thread 2!
In main: creating thread 4
Hello World from thread 3!
Hello World from thread 4!
```

Пример - вычисление определенного интеграла



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

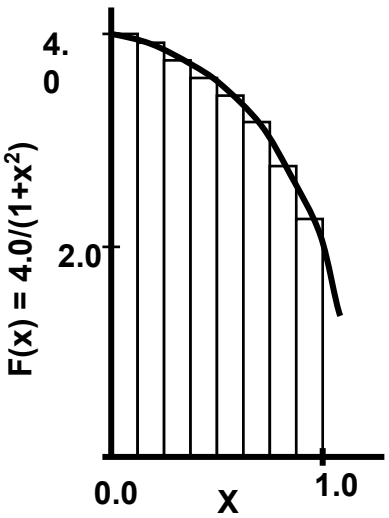
Аппроксимация по методу прямоугольников:

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

Каждый прямоугольник имеет ширину Δx и высоту $F(x_i)$ в середине интервала.

Вариант последовательной программы

```
PROGRAM PI_SERIAL

INTEGER I, N
DOUBLE PRECISION X, STEP, PI, SUM /0.0/
PARAMETER (N=100000000)
STEP = 1.0/N

DO I=1, N
  X = (I-0.5)*STEP
  SUM = SUM + 4.0/(1.0+X*X)
END DO

PI = STEP * SUM
PRINT *, 'PI = ', PI
END
```



Результаты выполнения программы:

```
% gfortran -o pi_serial pi_serial.f
% ./pi_serial
PI=3.1415926319813576
```

Вариант программы с использованием OpenMP

```
          PROGRAM PI_OPENMP
          INCLUDE 'omp_lib.h'
          INTEGER I, N, NTHREADS, TID, ISTART, IEND
          DOUBLE PRECISION X, STEP, PI, THREAD_SUM /0.0/
          PARAMETER (NUM_THREADS=2, N=100000000)
          STEP = 1.0/N

!$OMP PARALLEL IF(N .GE. 100) NUM_THREADS(NUM_THREADS)
!$OMP&  PRIVATE(TID, X, I, ISTART, IEND) REDUCTION(+:THREAD_SUM)
           TID = OMP_GET_THREAD_NUM()
           ISTART = TID+1
           IEND = N
           DO I = ISTART, IEND, NUM_THREADS
              X = (I-0.5)*STEP
              THREAD_SUM = THREAD_SUM + 4.0/(1.0+X*X)
           END DO
!$OMP END PARALLEL

           PI = STEP * THREAD_SUM
           PRINT *, 'PI = ', PI
           END
```

```
% gfortran -fopenmp -o pi_openmp pi_openmp.f
% export OMP_NUM_THREADS=2
% ./pi_openmp
PI=3.1415926319813576
```

Технология OpenMP ориентирована на применение для многопроцессорных систем с общей памятью (SMP-системы), вычислительные единицы - параллельно выполняющиеся потоки (нити).

Вариант программы с использованием MPI

Технология MPI ориентирована на применение для многопроцессорных систем с распределенной памятью (кластеры), вычислительные единицы - параллельно выполняющиеся процессы.

```
 PROGRAM PI_MPI
 INCLUDE 'mpif.h'
 DOUBLE PRECISION PROC_SUM, PI, H, SUM, X, F, A
 INTEGER N, PID, NUM_PROCS, I, IERR
 F(A) = 4. / (1.0 + A*A)

 CALL MPI_INIT(IERR)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, PID, IERR)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NUM_PROCS, IERR)
 IF ( PID .EQ. 0 ) THEN
    PRINT *, 'ENTER THE NUMBER OF INTERVALS: (0 QUITS)'
    READ(*,*) N
 ENDIF
 CALL MPI_BCAST(N,1,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
 H = 1.0/N
 SUM  = 0.0
 DO I = PID+1, N, NUM_PROCS
    X = H * (I - 0.5)
    PROC_SUM = PROC_SUM + F(X)
 ENDDO
 PROC_SUM = H * SUM
 CALL MPI_REDUCE(PROC_SUM,PI,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
&                MPI_COMM_WORLD,IERR)
 IF (PID .EQ. 0) THEN
    PRINT *, 'PI = ', PI
 ENDIF
 CALL MPI_FINALIZE(IERR)

 END
```

Компиляция и выполнение программы:

```
% mpif77 -o pi_mpi pi_mpi.f
% mpirun -np 2 ./pi_mpi
PI=3.1415926319813576
```

Вариант программы с использованием Win API

Используется низкоуровневая реализация механизмов многопоточности из Win API.

```
#include <stdio.h>
#include <windows.h>
#define NUM_THREADS 2
CRITICAL_SECTION hCriticalSection;
double pi = 0.0;
int n =100000;
void main () {
   int i, threadArg[NUM_THREADS];
   DWORD threadID;
   HANDLE threadHandles[NUM_THREADS];
   for(i=0; i<NUM_THREADS; i++)  threadArg[i] = i+1;
   InitializeCriticalSection(&hCriticalSection);
   for (i=0; i<NUM_THREADS; i++) threadHandles[i] =CreateThread(0, 0,
    (LPTHREAD_START_ROUTINE) Pi, &threadArg[i], 0, &threadID);
   WaitForMultipleObjects(NUM_THREADS, threadHandles, TRUE,INFINITE);
   printf("PI = %.16f", pi);
}
void Pi (void *arg) {
   int i, start;
   double h, sum, x;
   h = 1.0 / (double) n;
   sum = 0.0;
   start = *(int *) arg;
   for (i=start; i<= n; i=i+NUM_THREADS) {
       x = h * ((double)i - 0.5);
       sum += (4.0 / (1.0 + x*x)); }
       EnterCriticalSection(&hCriticalSection);
         pi += h * sum;
       LeaveCriticalSection(&hCriticalSection); }
   }
```

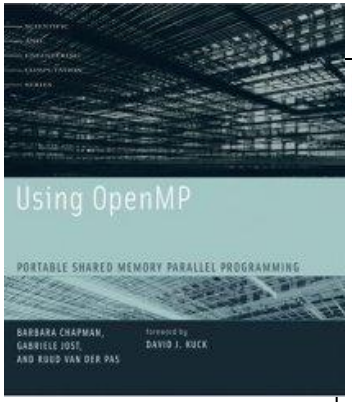
Технология OpenMP: общие сведения

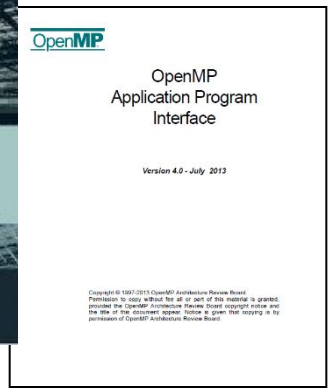
<u>OpenMP</u> (*Open Multi-Processing*) - развитый высокоуровневый интерфейс прикладного программирования (Application Programming Interface, API), предназначенный для параллелизации программ на многопроцессорных компьютерах с общим полем оперативной памяти.

<u>OpenMP</u> разработан для языков программирования ***Fortran*** и ***C/C++*** и может одинаково успешно использоваться при работе на операционных платформах UNIX (Linux) и MS Windows.

<u>Основные источники информации в Интернете:</u>

- OpenMP.org - *http://www.openmp.org*
- cOMPunity (Community of OpenMP Users, Researchers, Tool Developers and Providers) - *http://www.compunity.org*
- OpenMP Tutorial - *http://computing.llnl.gov/tutorials/openMP/*
- ИНТУИТ.ру – *http://www.intuit.ru (курсы по OpenMP)*





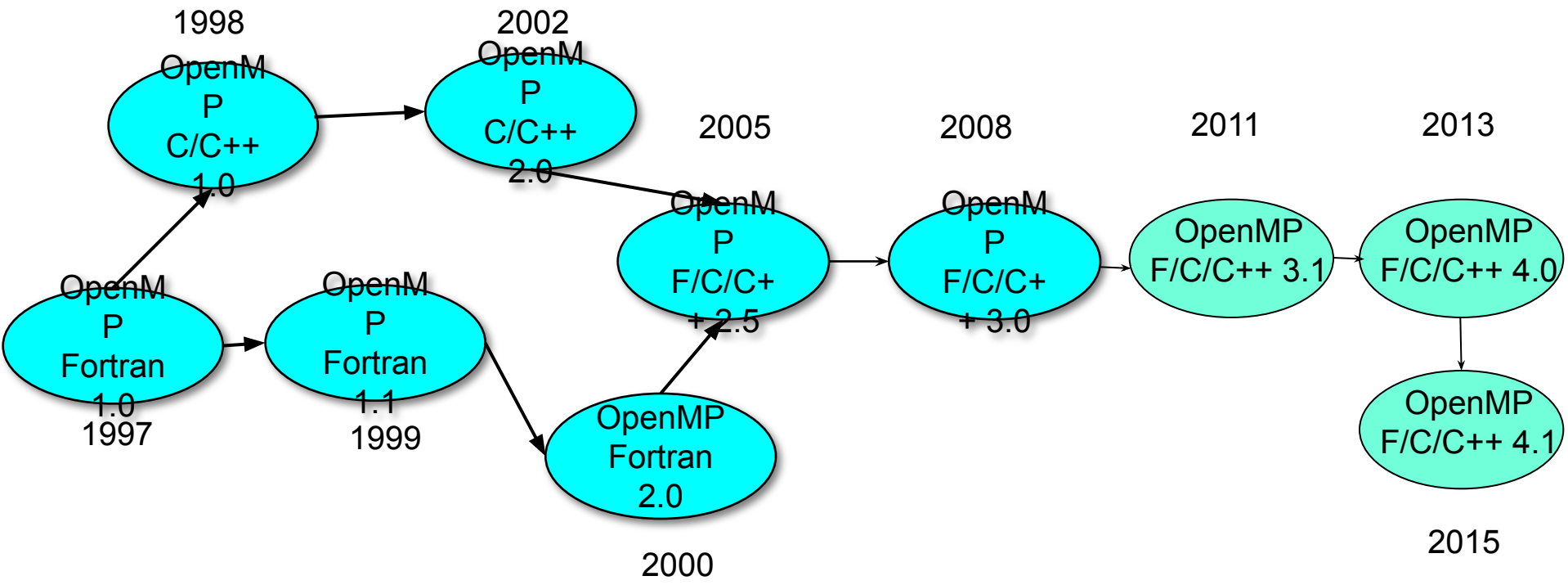
<u>Учебные пособия:</u>

Абрамов А.Г. Вычисления на многопроцессорных компьютерах. Параллельные вычисления на основе технологии OpenMP: Учебное пособие. - СПб.: Изд-во Политехн. ун-та, 2012. - 150 с.

Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. - М.: Изд-во МГУ, 2009. - 77 с. (электронная версия: http://parallel.ru/info/parallel/openmp)

Левин М.П. Параллельное программирование с использованием OpenMP. - М.: ИНТУИТ.ру, БИНОМ. Лаборатория знаний, 2008. - 120 с.

Версии спецификации OpenMP



http://www.openmp.org

Компиляторы с поддержкой OpenMP

Коммерческие компиляторы:

- **Intel C/C++ / Fortran Compilers** (*http://software.intel.com/en-us/intel-compilers/*)
- Microsoft Visual Studio (*https://www.visualstudio.com,* нет компилятора Fortran)
- The Portland Group (PGI) C/C++ / Fortran Compilers (*http://www.pgroup.com*)
- IBM XL C/C++ / Fortran Compilers (*http://www.ibm.com/software/awdtools/xlcpp/*, *http://www.ibm.com/software/awdtools/fortran/xlfortran/*)
- Oracle Solaris Studio (*http://www.oracle.com/technetwork/server-storage/solarisstudio/*)
- HP C/C++ / Fortran Compilers (*http://www.hp.com*)
- Lahey/Fujitsu Fortran Compiler (*http://www.lahey.com*)

Свободно распространяемые компиляторы:

- **GNU Compiler Collection** (начиная с версии 4.4, *http://gcc.gnu.org*)
- OpenUH Compiler Suite (*http://www.cs.uh.edu/~openuh/*)
- The Mercurium C/C++ / Fortran Compiler (NANOS project, *http://nanos.ac.upc.edu*)
- OMPi C Compiler (*http://www.cs.uoi.gr/~ompi/*)

Примеры компиляции OpenMP-программ на языках Fortran и C разными компиляторами

Компилятор GNU:

```
% gfortran -o omp_hello -fopenmp omp_hello.f
% gcc -o omp_hello -fopenmp omp_hello.c
```

Компилятор Intel:

```
% ifort -o omp_hello -openmp omp_hello.f
% icc -o omp_hello -openmp omp_hello.c
```

Основы OpenMP. Модель исполнения программ

Модель исполнения программ в OpenMP основана на параллельной работе разделяющих общую память нитей (потоков), находящихся в ведении управляющей системы стандарта (runtime system). Используемая концепция многопоточного программирования ориентирована на создание, выполнение и размещение потоков в программе, а также на координацию потоков, выполняемых ими операций и используемых ресурсов.

Процесс с позиций UNIX-подобных операционных систем ассоциируется с:

- последовательным набором исполняемых им операций (потоком команд);
- задействованных в операциях данных;
- используемых ресурсов (оперативная память, файлы, устройства ввода-вывода и т.д.);
- текущим состоянием (значение программного счетчика, указатель стека, содержимое регистров)

Процессы надежно изолированы друг от друга, каждый их них работает в собственной, *независимой области памяти* (виртуальном адресном пространстве), а для организации взаимодействия процессов применяются специальные *механизмы межпроцессных коммуникаций*.

Процесс может состоять из одной или нескольких ***нитей (thread)*** или ***потоков исполнения***, разделяющих между собой его ресурсы (адресное пространство, программный код, файлы и др.). Каждая ***нить***, обладая собственным программным счетчиком, регистровым контекстом и стеком, может выполнять независимый набор операций в программе.

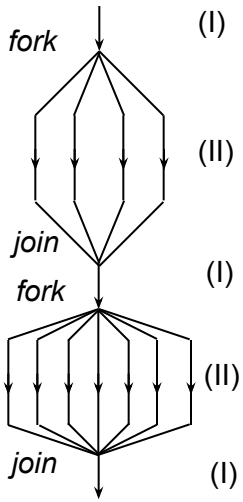
Разделение в рамках одного процесса и совместное использование нитями общих ресурсов снижает "накладные расходы" на операции по созданию и управлению нитями, по сравнению с имеющими место при работе с процессами. Взаимодействие нитей осуществляется напрямую, через общую разделяемую память - отсутствует необходимость организовывать дополнительные дорогостоящие пересылки данных, требуемые при совместной работе процессов.

Наиболее важной задачей при разработке многопоточных приложений является обеспечение корректной и эффективной ***синхронизированной работы нитей*** в условиях конкурентного использования общих для них ресурсов.

Основы OpenMP. Модель исполнения программ

OpenMP базируется на *высокоуровневой* реализации механизмов многопоточности, поддерживаемых системой управления стандарта.

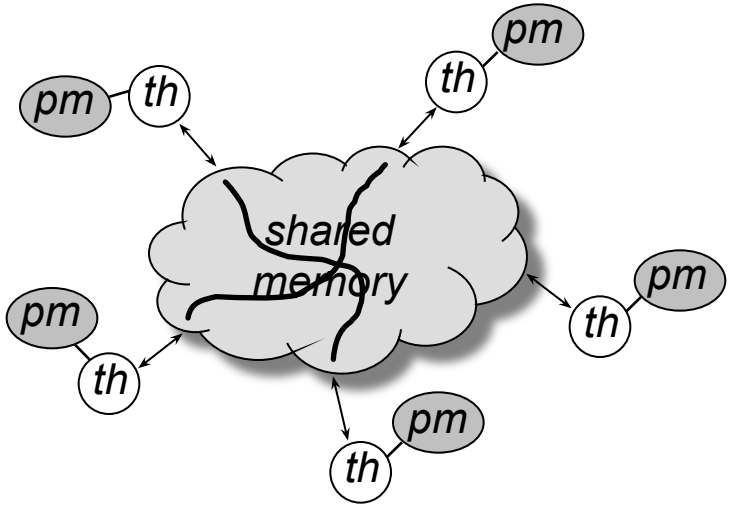
Используется так называемая ***"fork-join"*** (***"вилочная"*** или ***"пульсирующая"***) модель выполнения распараллеленной программы, основанная на порождении и объединении нитей в процессе ее выполнения.



- OpenMP-программа структурно состоит из набора следующих друг за другом последовательных и специальным образом помеченных параллельных областей кода
- в момент запуска программы создается единственная нить (главная нить или нить-мастер), которая единолично выполняет все последовательные участки кода
- вход в параллельную область сопровождается порождением фиксированного числа дополнительных нитей (операция ***fork***), которые вместе с главной формируют группу (команду, *team*) нитей для исполнения в параллельном режиме соответствующего этой области программного кода
- нитям с целью их однозначной идентификации в пределах группы присваиваются уникальные номера (последовательные натуральные числа), нулевой номер всегда зарезервирован за главной нитью
- поскольку нить обладает собственным стеком команд, каждой из них в пределах параллельной области может быть назначен для выполнения индивидуальный набор операций, для чего в API OpenMP имеется набор конструкций распределения работы
- привязка нитей к разным процессорам (ядрам) осуществляется системой поддержки исполнения стандарта (типичный режим назначения - одна нить на одно ядро)
- перед выходом из параллельной области главная нить дожидается завершения выполнения операций, закрепленных за дополнительными нитями из группы, после чего происходит уничтожение этих нитей (операция ***join***)

Основы OpenMP. Модель памяти

Модель памяти, на которую опирается стандарт OpenMP, предполагает, что все нити имеют равноправные возможности доступа к глобально разделяемой памяти (*uniform memory access, UMA*).



Переменные в параллельных областях OpenMP-программы могут принадлежать к одному из двух ***классов***:

- **общие** (разделяемые, *shared*)
- **локальные** (частные, *private*)

Общие переменные существуют в программе в единственном экземпляре и доступны всем нитям под одним и тем же именем (по одному и тому же физическому адресу в памяти). Выполнение операций чтения/записи с общими переменными представляет собой прозрачный механизм организации взаимодействия нитей.

Локальные переменные создаются внутри параллельной области в виде независимых копий общих переменных, никак, помимо одинакового имени, с ними не связанных. Локальные переменные доступны для использования нитью-владельцем только в пределах параллельной области (уничтожаются при выходе из нее) и преимущественно используются для временного хранения результатов вычислений.

Каждая нить, кроме того, имеет в своем единоличном распоряжении специальную область памяти (называемую *threadprivate memory*), предназначенную для хранения локальных переменных, существующих на протяжении всей программы.

Основы OpenMP. Модель памяти

Одна из важнейших задач при разработке многопоточных приложений - обеспечение корректной и эффективной синхронизированной работы нитей в условиях конкурентного использования общей оперативной памяти.

Методы обеспечения синхронизации:

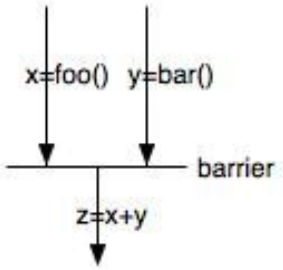
- взаимное исключение (*mutex*, *mutual exclusion*)
- синхронизация по событию (*event synchronization*).

Семафор - объект в программировании , позволяющий одновременно войти в заданный участок кода не более чем n нитям. *Взаимное исключение* - одноместный семафор (n=1), служащий для синхронизации одновременно выполняющихся нитей.

Формулировка задачи *взаимного исключения*: необходимо согласовать работу параллельных процессов (нитей) при использовании критического ресурса (разделяемой памяти) таким образом, чтобы удовлетворить следующим основным требованиям:

- одновременно внутри критической области должно находиться не более одной нити;
- критические области не должны иметь приоритета в отношении друг друга;
- остановка какой-либо нити вне его критической области не должна влиять на дальнейшую работу нитей по использованию критического ресурса;
- освобождение критического ресурса и выход из критической области должны быть произведены нитью, использующей критический ресурс, за конечное время.

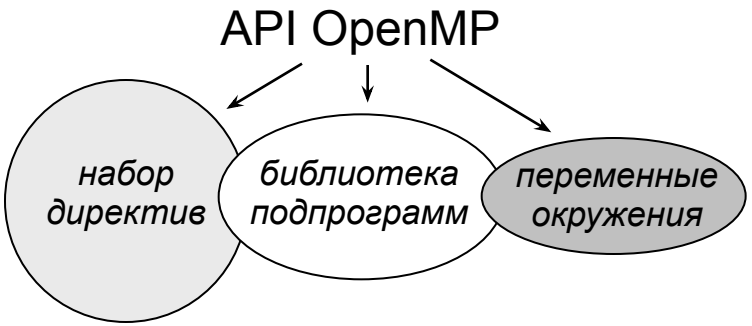
Синхронизация по событиям - параллельно выполняющиеся нити приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другая нить. Наиболее простой формой такой синхронизации является *барьер*.



Интерфейс прикладного программирования (API) OpenMP

Интерфейс прикладного программирования OpenMP реализуется с помощью трех базовых компонент стандарта:

- директив компилятора (*compiler directives*)
- библиотечных подпрограмм времени исполнения (*runtime library routines*)
- переменных окружения (*environment variables*)



За низкоуровневые детали реализации отвечает компилятор и управляющая система стандарта, а разработчик занимается выбором и указанием в программе наиболее эффективных режимов параллельного выполнения и синхронизации нитей.

Директивы OpenMP служат для указания компилятору требующих параллельного выполнения частей программы, распределения работы между нитями в параллельных областях, а также для координации доступа нитей к общей памяти.

Библиотечные подпрограммы и ***переменные окружения*** OpenMP (частично дублирующие функции друг друга) позволяют задавать и осуществлять контроль над параметрами среды исполнения программы применительно к особенностям обработки параллельных областей (количество порождаемых нитей, способ распределения между нитями итераций в распараллеленных циклах и др.).

Общие сведения о директивах OpenMP

Термин ***директива*** (*directive*) закреплен в OpenMP за наиболее многочисленной конструкцией стандарта, позволяющей реализовать в программе основной функционал этой технологии параллельного программирования.

Классификация директив по типу выполняемых действий:

- директива назначения параллельных областей программы (*parallel construct*);
- директивы распределения работы между нитями в параллельных областях (*work-sharing constructs*);
- директивы синхронизации (*synchronization constructs*).

Большинство директив OpenMP являются исполняемыми (*executable*), могут либо иметь, либо не иметь ассоциированный с ними программный код. Область действия исполняемой директивы распространяется на часть программы, называемой в стандарте *структурным блоком* (*structured block*) - фрагмент кода (одиночный оператор или последовательность операторов) с одной точкой входа (в начале) и одной точкой выхода (в конце).

Директивы OpenMP могут сопровождаться *атрибутами* (или опциями, *clause*), которые позволяют определять дополнительные свойства соответствующих директив, актуальные на протяжении их выполнения. Атрибуты могут задавать специфичные для OpenMP классы переменных, число нитей в параллельной области программы, особенности выполнения распараллеленных алгоритмических конструкций (например, циклов) и т.д.

Общие сведения о директивах OpenMP: синтаксис директив

Общие синтаксические правила включения директив и атрибутов OpenMP в исходные коды программ для языков стандарта - Fortran и C/C++:

Язык Fortran

```
!$OMP DIRECTIVE-NAME [CLAUSE[ [,] CLAUSE]...]
C$OMP DIRECTIVE-NAME [CLAUSE[ [,] CLAUSE]...]
*$OMP DIRECTIVE-NAME [CLAUSE[ [,] CLAUSE]...]
```

Примеры:

```
!$OMP PARALLEL PRIVATE(TID, A) SHARED(NTHREADS)
!$OMP DO REDUCTION(+:SUM)
```

Язык C/C++

```
#pragma omp directive-name [clause[ [,] clause]...]
```

Примеры:

```
#pragma omp parallel private(tid, a) shared(nthreads)
#pragma omp for reduction(+:sum)
```

Директивы OpenMP в программах на языке Fortran располагаются в комментариях специального формата, а в C/C++ для их записи используется предусмотренный в стандарте этих языков механизм команд препроцессора (директив компилятора).

Существуют особенности оформления директив в зависимости от используемого способа записи языковых элементов языка Fortan (фиксированная, свободная).

Общие сведения о директивах OpenMP: условная компиляция программ

Организация ***условной компиляции*** программы позволяет развивать единый код программы для параллельного и последовательного исполнения.

Fortran: помещение конструкций OpenMP и операторов, не требующих исполнения в случае отсутствия поддержки OpenMP компилятором, в строки, помеченные символами комментария специального вида.

C/C++: использование специального макроса с именем _OPENMP, в значении которого зашифрован год и месяц поддерживаемого соответствующим компилятором версии стандарта OpenMP (в формате yyyymm).

В случае последовательного выполнения и использования компилятора, не имеющего поддержки OpenMP, директивы просто будут игнорироваться, а для библиотечных подпрограмм можно применять процедуры-заглушки (stubs), текст которых приведен в спецификации стандарта.

```
   PROGRAM OPENMP_SUPPORTED
C THIS IS A REAL FORTRAN COMMENT
!$ PRINT *, 'Compiled by an OpenMP-compliant implementation'
   END
```

```
#include <stdio.h>
int main (int argc, char *argv[]) {
#ifdef _OPENMP
 printf("OpenMP is supported. Version of specification: %d\n", _OPENMP);
#else
 printf("OpenMP is not supported\n");
#endif
return 0;
}
```

```
%./omp_supported
OpenMP is supported. Version of specification:
200805
```

Общие сведения о библиотечных подпрограммах

Библиотечные подпрограммы OpenMP позволяют задавать и осуществлять контроль над параметрами среды исполнения программы применительно к особенностям обработки параллельных областей (количество порождаемых нитей, способ распределения между нитями итераций в распараллеленных циклах и др.).

Возможность вызова библиотечных подпрограмм OpenMP обеспечивается при явном подключении в коде программы имени соответствующей библиотеки (заголовочного файла):

<u>Язык Fortran</u>

```
INCLUDE 'omp_lib.h'
```

<u>Язык C/C++</u>

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

Применение условной компиляции программ при использовании библиотечных подпрограмм:

```
!$   N = OMP_GET_NUM_THREADS()
```

В описании стандарта приведены коды ***подпрограмм-"заглушек"*** на языках Fortran и C/C++, которыми можно подменить обращения к реальным библиотечным подпрограммам при использовании компиляторов без поддержки OpenMP.

```
  INTEGER FUNCTION OMP_GET_NUM_THREADS()
    OMP_GET_NUM_THREADS = 1
    RETURN
  END FUNCTION
```

Общие сведения о переменных окружения

Переменные окружения OpenMP служат для создания переносимой среды запуска параллельных программ. Использование переменных окружения позволяет контролировать поведение программы, обходясь без ее перекомпиляции, в случаях, когда требуется изменить количество нитей, выполняющих параллельные области программы, задать тот или иной способ распределения итераций между нитями в распараллеленных циклах, разрешить или запретить вложенный параллелизм и т.д.

Переменные окружения в значительной степени дублируют функционал библиотечных подпрограмм времени исполнения OpenMP. Библиотечные подпрограммы OpenMP при одновременном использовании имеют приоритет над переменными окружения.

Установка переменных окружения в UNIX-подобных операционных системах (команды *export* в ksh и bash, *setenv* - в csh):

```
% export OMP_NUM_THREADS=4
% setenv OMP_NUM_THREADS 4

% echo $OMP_NUM_THREADS
4
```

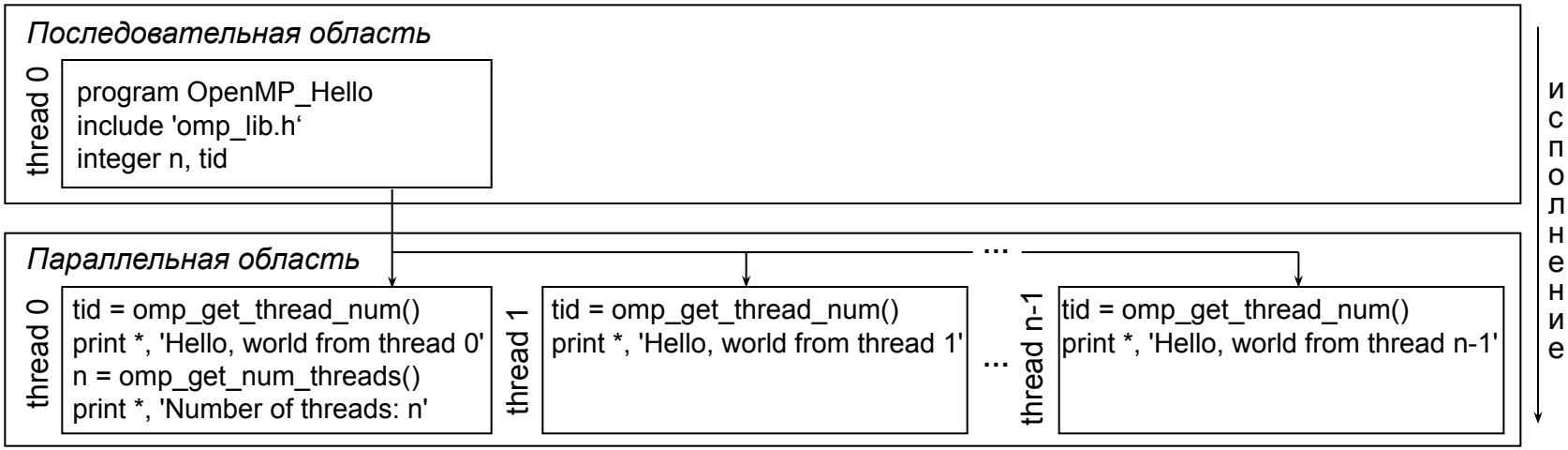
В операционных системах семейства MS Windows для установки и просмотра значений переменных окружения в режиме командной строки применяются команды *set* и *echo*:

```
> set OMP_NUM_THREADS=4
> echo %OMP_NUM_THREADS%
4
```

Пример параллельной программы: “Hello, world!”

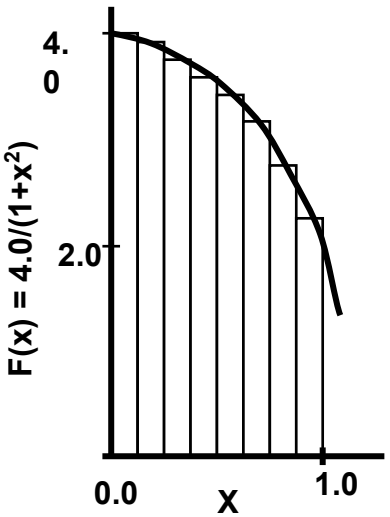
```
PROGRAM OPENMP_HELLO_WORLD
      INCLUDE 'omp_lib.h'
      INTEGER N, TID
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello, world from thread ', TID
      IF (TID .EQ. 0) THEN
        N = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads: ', N
      END IF
!$OMP END PARALLEL
      END
```

```
% gfortran -fopenmp -o omp_hello omp_hello.f
% export OMP_NUM_THREADS=4
% ./omp_hello
Hello, world from thread 0
Number of threads: 4
Hello, world from thread 2
Hello, world from thread 3
Hello, world from thread 1
```



Пример параллельной программы: вычисление определенного интеграла

```
      PROGRAM OPENMP_PI
      INCLUDE 'omp_lib.h'
      INTEGER I, N, NTHREADS, TID, ISTART, IEND
      DOUBLE PRECISION X, STEP, PI, THREAD_SUM /0.0/
      PARAMETER (NTHREADS=2, N=100000000)
      STEP = 1.0/N

!$OMP PARALLEL IF(N .GE. 100) NUM_THREADS(NTHREADS)
!$OMP&  PRIVATE(TID, X, I, ISTART, IEND) REDUCTION(+:THREAD_SUM)
      TID = OMP_GET_THREAD_NUM()
      ISTART = TID+1
      IEND = N
      DO I=ISTART, IEND, NTHREADS
         X = (I-0.5)*STEP
         THREAD_SUM = THREAD_SUM + 4.0/(1.0+X*X)
      END DO
!$OMP END PARALLEL

      PI = STEP * THREAD_SUM
      PRINT *, 'PI = ', PI
      END
```




```
% gfortran -fopenmp -o pi_openmp pi_openmp.f
% export OMP_NUM_THREADS=2
% ./pi_openmp
PI=3.1415926319813576
```

Директивы и атрибуты OpenMP

Директивы и атрибуты OpenMP.
Назначение параллельных областей программы

Язык Fortran

```
!$OMP PARALLEL [CLAUSE[[,] CLAUSE]...]
  STRUCTURED-BLOCK
!$OMP END PARALLEL
```

Язык C/C++

```
#pragma omp parallel [clause[[,] clause]...]
  structured-block
```

Возможные атрибуты директивы `PARALLEL`:

- спецификация класса для используемых в параллельной области (структурном блоке) переменных (`PRIVATE, SHARED, FIRSTPRIVATE, REDUCTION, COPYIN`)
- `DEFAULT(TYPE)` - задание класса переменных "по умолчанию", либо указание на необходимость явного назначения класса для всех используемых в параллельной области переменных
- `NUM_THREADS(SCALAR-INTEGER-EXPRESSION)`- задание числа нитей в параллельной области
- `IF(SCALAR-LOGICAL-EXPRESSION)`- параллелизация структурного блока по условию
- `PROC_BIND(MASTER | CLOSE | SPREAD)` - контроль привязки нитей к ядрам

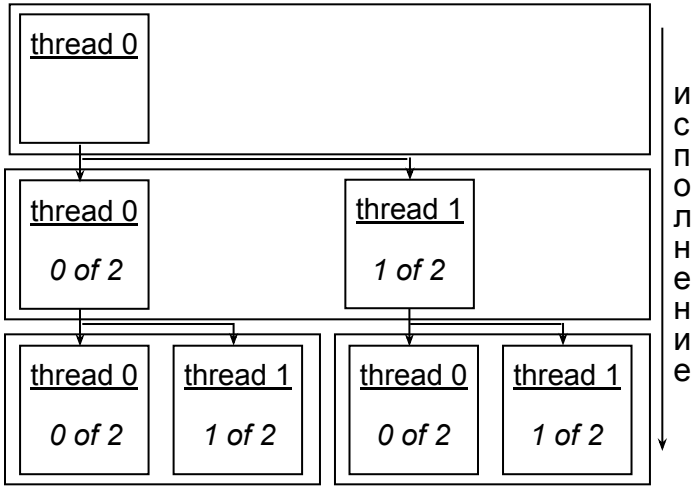
Многоуровневый (вложенный) параллелизм

Стандарт OpenMP разрешает использовать в программах неограниченную вложенность параллельных областей друг в друга. Статус *многоуровневого (вложенного) параллелизма* (*nested parallelism*) устанавливается и контролируется соответствующими библиотечными подпрограммами и переменными окружения OpenMP.

В зависимости от статуса вложенные параллельные области либо:

- будут исполняться только одной нитью из группы (которая первой достигнет точки входа в область)
- для исполнения вложенных областей могут порождаться дополнительные нити

```
 PROGRAM OPENMP_NESTED
      INCLUDE 'omp_lib.h'
      INTEGER N, TID
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      IF (TID .EQ. 0) THEN
        N = OMP_GET_NUM_THREADS()
      END IF
      PRINT *, 'Outer parallel region: thread ', TID,
     & ' of ', N
      CALL OMP_SET_NESTED(.TRUE.)
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      IF (TID .EQ. 0) THEN
        N = OMP_GET_NUM_THREADS()
      END IF
      PRINT *, 'Inner parallel region: thread ', TID,
     & ' of ', N
!$OMP END PARALLEL
!$OMP END PARALLEL
      END
```




```
% export OMP_NUM_THREADS=2
% ./omp_nested
Outer parallel region: thread 0 of 2
Outer parallel region: thread 1 of 2
Inner parallel region: thread 0 of 2
Inner parallel region: thread 1 of 2
Inner parallel region: thread 0 of 2
Inner parallel region: thread 1 of 2
```

Классы переменных и управление данными

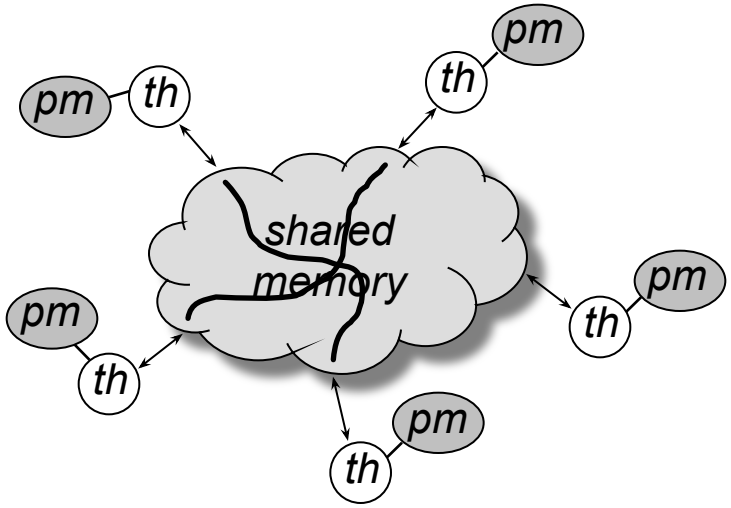
Переменные в параллельных областях OpenMP-программы (с позиций доступности для обращения к ним со стороны нитей) могут принадлежать к одному из двух основных классов:

- ***общие*** (разделяемые, ***shared***)
- ***локальные*** (приватные, ***private***)

Дополнительные правила управления данными регламентируют поведение переменных при входе и выходе из параллельной области или из распараллеленного средствами OpenMP цикла.

Соответствующие классы переменных и управляющие конструкции задаются с помощью специального набора *атрибутов совместного использования данных*.

Атрибуты задания общих и локальных переменных



`SHARED(`*`LIST`*`)`
`PRIVATE(`*`LIST`*`)`

Атрибут `SHARED` задает список общих (разделяемых) нитями переменных, каждая из которых доступна всем нитям под одним и тем же именем и по одному и тому же адресу в оперативной памяти.

Атрибут `PRIVATE` служит для назначения локальных (приватных) по отношению к нитям переменных. Для каждой из нитей при входе в параллельную область создается отдельный, независимый экземпляр (копия) локальной переменной с сохранением типа оригинальной общей переменной. Локальные переменные не имеют никакой связи с оригинальной переменной и уничтожаются при выходе из параллельной области. Значения локальных переменных являются не инициализированными при входе в параллельную область и неопределенными при ее завершении.

По умолчанию большинство переменных в OpenMP-программе считаются *общими*. Исключения:

- переменные, впервые объявленных внутри параллельной области
- счетчики итераций циклов, распараллеленных с помощью соответствующих OpenMP-директив

Классы переменных в OpenMP

Инициализация и сохранение значений локальных переменных

FIRSTPRIVATE(*LIST*)
LASTPRIVATE(*LIST*)

Атрибут-оператор FIRSTPRIVATE позволяет выполнить автоматическую инициализацию создаваемых в нитях копий локальных переменных. Для переменных, перечисленных в списке параметров данного атрибута, значения созданных локальных копий во всех нитях будут проинициализированы значением оригинальной переменной из главной нити, актуальным на момент входа в параллельную область.

Атрибут-оператор LASTPRIVATE позволяет сохранить и сделать доступным за пределами параллельной области последнее по времени модификации значение локальной переменной. Для соответствующих переменных по завершении параллельной области происходит обновление значения оригинальной, общей переменной значением, записанным последним той или иной нитью в собственную локальную переменную внутри параллельной области.

```
 PROGRAM OPENMP_FIRSTPRIVATE
      INCLUDE 'omp_lib.h'
      INTEGER TID /-1/
!$OMP PARALLEL FIRSTPRIVATE(TID)
      PRINT *, 'Initial value: ', TID
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Thread number: ', TID
!$OMP END PARALLEL
      PRINT *, 'Final value: ', TID
      END
```

Результат выполнения программы:

```
% export OMP_NUM_THREADS=3
% ./omp_firstprivate
Initial value: -1
Thread number: 2
Initial value: -1
Thread number: 1
Initial value: -1
Thread number: 0
Final value: -1
```

Задание класса переменных по умолчанию

DEFAULT(*TYPE*)

Атрибут DEFAULT служит для явного назначения класса переменных, используемого в режиме "по умолчанию" (указанием в качестве параметра вызова названия класса). Установка для атрибута значения NONE означает, что класс по умолчанию в параллельной области не определен и требуется его явное задание для всех задействованных в ней переменных.

Классы переменных в OpenMP

Совместная обработка локальных переменных

Специальный тип переменных, задаваемый атрибутом-оператором `REDUCTION`, объединяет в себе черты общих и локальных переменных OpenMP и применяется к переменным, с которыми в параллельной области необходимо выполнить некоторую операцию совместной обработки (приведения или *редукции*). Использование редукции является предпочтительным и наиболее эффективным способом решения задачи по разграничению доступа нитей к общим переменным.

`REDUCTION({`*`OPERATOR`*` |`*`INTRINSIC_PROCEDURE_NAME`*`}:`*`LIST`*`)`

Для отнесенной к типу `REDUCTION` переменной создается ее локальная копия в каждой из нитей, с последующей ее инициализацией, выполняемой с учетом функционала применяемого оператора или процедуры.

Возможные действия (*`OPERATOR`*) :

- арифметические операции `(+, *, -)`
- логические операторы `(.AND., .OR., .EQV., .NEQV., C/C++ - &, |, ^, &&, ||)`
- встроенные процедуры: `MAX, MIN, IAND, IOR, IEOR`

```
 PROGRAM OPENMP_REDUCTION
      INTEGER N /0/
!$OMP PARALLEL REDUCTION(+:N)
      N=N+1
!$OMP END PARALLEL
      PRINT *, 'Number of threads: ', N
      END
```

Результат выполнения программы:

```
% export OMP_NUM_THREADS=4
% ./omp_reduction
Number of threads: 4
```

Управление данными в OpenMP

Назначение постоянно существующих в программе локальных переменных

Описательная (декларативная) директива `THREADPRIVATE` позволяет сделать глобальные переменные в программе локальными по отношению к OpenMP-нитям, при этом такие переменные будут существовать и сохранять свои значения на всем протяжении работы программы.

`!$OMP THREADPRIVATE(`*`LIST`*`)`

`LIST` - разделенный запятыми список глобальных переменных (C/C++) или именованных структурных блоков (Fortran).

<u>Особенности работы директивы:</u>

- до первой параллельной области работает только одна главная нить, для которой и создаются локальные копии объявленных *threadprivate*-переменных
- при входе в параллельную область порождаются дополнительные нити, и происходит репликация (размножение) заданных в списке параметров директивы `THREADPRIVATE` исходно общих переменных
- начальная инициализация копий *threadprivate*-переменных производится в соответствии с правилами используемого языка программирования так, как бы это делалось в случае последовательного выполнения программы
- завершение параллельной области сопровождается "исчезновением", приостановкой работы всех нитей, за исключением главной
- приостановленные, но не "завершенные" нити будут находиться в ожидании следующей параллельной области, имея при этом возможность сохранять в специальной части памяти и передавать значения *threadprivate*-переменных между разными параллельными областями программы
- динамическое изменение количества нитей в программе должно быть запрещено (все параллельные области должны обрабатываться одинаковым числом нитей)

Управление данными в OpenMP

Назначение постоянно существующих в программе локальных переменных

```
      PROGRAM OPENMP_THREADPRIVATE
      INCLUDE 'omp_lib.h'
      INTEGER I, TID
      COMMON /BLOCK/ I
!$OMP THREADPRIVATE(/BLOCK/)
      I=-1
      CALL OMP_SET_DYNAMIC(.FALSE.)
      PRINT *, 'First parallel region'
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Thread number: ', TID ,', I=',I
      I = TID
      PRINT *, 'Thread number: ', TID ,', I=',I
!$OMP END PARALLEL
      PRINT *, 'Serial region: I=',I
      PRINT *, 'Second parallel region'
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Thread number: ', TID ,', I=',I
!$OMP END PARALLEL
      END
```

Результат выполнения программы:

```
% export OMP_NUM_THREADS=2
% ./omp_threadprivate
First parallel region
Thread number: 0, I=-1
Thread number: 1, I=0
Thread number: 0, I=0
Thread number: 1, I=1
Serial region:    I=0
Second parallel region
Thread number: 1, I=1
Thread number: 0, I=0
```

Переменная `I` объявлена как `THREADPRIVATE` и сохраняет значения для каждой из нитей (равное ее номеру) при переходе в программе между двумя параллельными областями. В первой параллельной области копия переменной `I` во второй нити инициализируется нулевым значением в соответствии с правилами языка Fortran.

Все переменные в Fortran имеют локальную область видимости - известны только в том модуле (главная программа, подпрограмма), в котором объявлены. Для возможности обращения к переменной из разных модулей, ее необходимо поместить в `COMMON` блок. `COMMON` блок может иметь имя, обладающее глобальной видимостью (известно во всех модулях); в программе может быть несколько `COMMON` блоков, которые должны иметь разные имена.

Управление данными в OpenMP

Атрибуты копирования значений локальных переменных

Атрибуты копирования значений переменных реализуют передачу, копирование локальных и *threadprivate*-переменных из одной нити в соответствующие переменные других нитей из группы

```
COPYIN(LIST)
COPYPRIVATE(LIST)
```

Применение атрибута-оператора `COPYIN` приводит к тому, что значение каждой переменной из списка *`LIST`*, предварительно объявленной как `THREADPRIVATE`, перед началом выполнения параллельной области устанавливается равным значению этой переменной в главной нити (атрибут-оператор инициирует копирование этого значения внутрь параллельной области - операция "*copy-in*").

Атрибут `COPYPRIVATE` реализует копирование значения локальной переменной из одной нити во все остальные нити из группы при нахождении непосредственно внутри параллельной области. Он может быть использован только совместно с директивой `SINGLE`.

При работе с большими массивами и выделении для них памяти с помощью процедуры `ALLOCATE()` соответствующие массивы в параллельных областях программы с точки зрения классов переменных OpenMP могут быть объявлены как: `PRIVATE`, `LASTPRIVATE`, `FIRSTPRIVATE`, `REDUCTION`, `COPYPRIVATE`.

```
 PROGRAM OPENMP_COPYIN
      INCLUDE 'omp_lib.h'
      INTEGER I, TID
      COMMON /BLOCK/ I
!$OMP THREADPRIVATE(/BLOCK/)
      I=-1
!$OMP PARALLEL COPYIN(I) PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Thread number: ', TID ,', I=',I
!$OMP END PARALLEL
      END
```

<u>Результат выполнения программы</u>:

```
% export OMP_NUM_THREADS=2
% ./omp_threadprivate
Thread number: 0, I=-1
Thread number: 1, I=-1
```

Распределение работы между нитями в параллельных областях

Явное распределение работы между нитями

Распределение выполняемых внутри параллельной области операций между нитями может быть организовано путем прямого назначения некоторого набора операторов конкретным нитям. Для этих целей необходимо задействовать библиотечные процедуры OpenMP, возвращающие номера нитей и (если потребуется) общее количество нитей в группе.

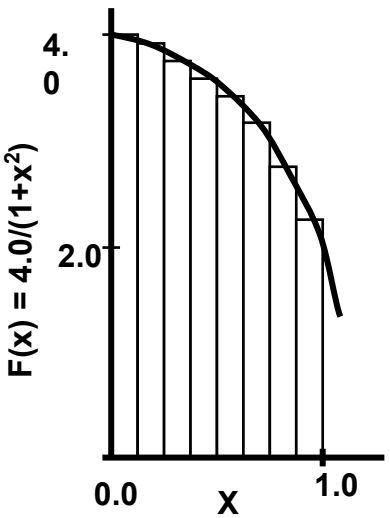
Подобная модель организации параллелизма в определенной степени соответствует применяемой при программировании на основе низкоуровневых библиотек многопоточности, реализующих стандарт POSIX.

```
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      IF (TID .EQ. 0) THEN
        CALL WORK1()
      ELSE IF (TID .EQ. 1) THEN
        CALL WORK2()
      ...
      END IF
!$OMP END PARALLEL
```

Данный способ уже использовался при написании программы вычисления определенного интеграла, где явным образом нитям назначались разные наборы обрабатываемых индексов цикла (и вычислялись площади соответствующих прямоугольников).

Явное распределение работы между нитями. Пример 1

```
 PROGRAM OPENMP_PI_SPMD
      INCLUDE 'omp_lib.h'
      INTEGER I, N, NTHREADS, TID, ISTART, IEND
      DOUBLE PRECISION X, STEP, PI, THREAD_SUM /0.0/
      PARAMETER (NTHREADS=2, N=100000000)
      STEP = 1.0/N
!$OMP PARALLEL IF(N .GE. 100) NUM_THREADS(NTHREADS)
!$OMP&  PRIVATE(TID, X, I, ISTART, IEND) REDUCTION(+:THREAD_SUM)
      TID = OMP_GET_THREAD_NUM()
      ISTART = TID+1
      IEND = N
      DO I = ISTART, IEND, NTHREADS
        X = (I-0.5)*STEP
        THREAD_SUM = THREAD_SUM + 4.0/(1.0+X*X)
      END DO
!$OMP END PARALLEL
      PI = STEP * THREAD_SUM
      PRINT *, 'PI = ', PI
      END
```

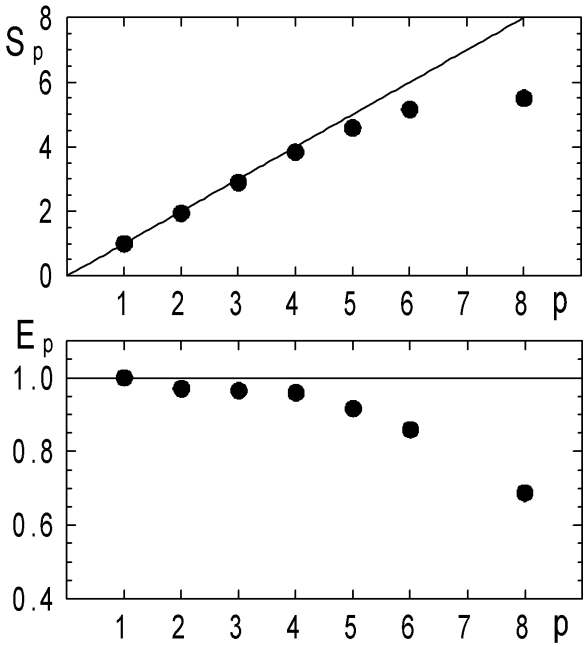


Результат выполнения программы:

```
% ./omp_pi_spmd
PI=3.1415926319813576
```

Явное распределение работы между нитями. Пример 2

```
 PROGRAM OPENMP_DAXPY_SPMD
      INCLUDE 'omp_lib.h'
      INTEGER I, ISTART, IEND, N, NTHREADS, TID
      PARAMETER (NTHREADS=2, N=100000000)
      DOUBLE PRECISION S /0.1/, T1, T2
      DOUBLE PRECISION, ALLOCATABLE :: Y(:), X(:)
      ALLOCATE (Y(N), X(N))
      X(1:N) = 1.0
      Y(1:N) = 2.0
      T1 = OMP_GET_WTIME()
!$OMP PARALLEL IF(N .GE. 100) NUM_THREADS(NTHREADS)
!$OMP&  PRIVATE(TID, I, ISTART, IEND)
      TID = OMP_GET_THREAD_NUM()
      ISTART = TID * N / NTHREADS + 1
      IEND = ISTART + N / NTHREADS - 1
      DO I=ISTART, IEND
        Y(I) = Y(I) + S*X(I)
      END DO
!$OMP END PARALLEL
      T2 = OMP_GET_WTIME()
      PRINT *, 'computational time: ', T2-T1, 's'
      DEALLOCATE (X, Y)
      END
```



Ускорение и эффективность распараллеливания (графики зависимости от числа нитей / ядер) в задаче о суммировании двух векторов (операция DAXPY в терминологии библиотеки BLAS).

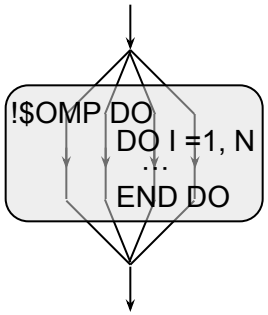
```
% time ./omp_daxpy_spmd
computational time: 0.853s
real 0m4.835s
user 0m3.744s
sys  0m1.732s
```

Суммарное время работы программы (`real`) в несколько раз превосходит затраченное непосредственно на вычисления в параллельной области (`computational time`), поскольку значительная часть времени тратится на начальную инициализацию векторов.

Директивы параллелизации циклов

Технология OpenMP имеет средства для применения подхода к распараллеливанию циклов, в значительной степени скрывающего от разработчика и передающего в ведение реализации стандарта конкретные детали разбиения цикла на независимые части и распределения этих частей между нитями. Такой высокоуровневый подход основывается на применении специальных *директив параллелизации циклов* (*parallel loop constructs*), которые помечают их для многопоточного исполнения.

```
!$OMP DO [CLAUSE[[,] CLAUSE] ... ]
  DO-LOOPS
[!$OMP END DO [NOWAIT] ]
```



<u>Возможные атрибуты:</u>

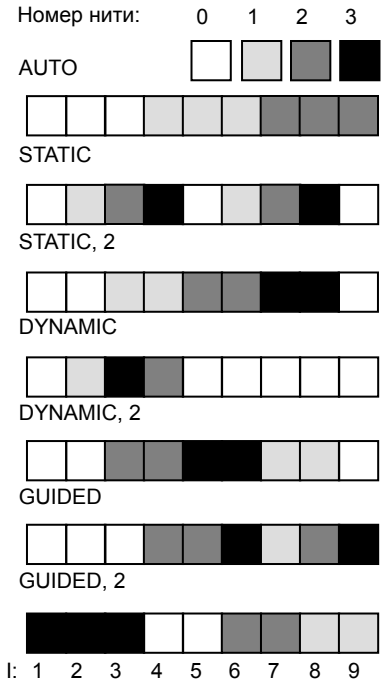
```
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
LASTPRIVATE(LIST)
REDUCTION(OPERATOR:LIST)
SCHEDULE(TYPE[, CHUNK_SIZE])
COLLAPSE(N)
ORDERED
```

Способы распределения итераций распараллеленного цикла между нитями, задаваемые атрибутом `SCHEDULE`

TYPE	**Описание**
STATIC	статический, блочно-циклический - распределение блоками по *CHUNK_SIZE* следующих подряд итераций (в порядке увеличения номеров нитей)
DYNAMIC	динамический - распределение блоками по *CHUNK_SIZE* следующих подряд итераций, при этом освободившаяся нить принимает к исполнению первый по порядку еще не обработанный блок
GUIDED	управляемый – динамическое распределение блоками следующих подряд итераций с постепенным уменьшением размера блока вплоть до величины *CHUNK_SIZE*
AUTO	способ распределения выбирается автоматически компилятором
RUNTIME	способ и размер блока выбирается во время выполнения программы путем установки соответствующего значения переменной окружения OMP_SCHEDULE или использования подпрограммы OMP_SET_SCHEDULE()

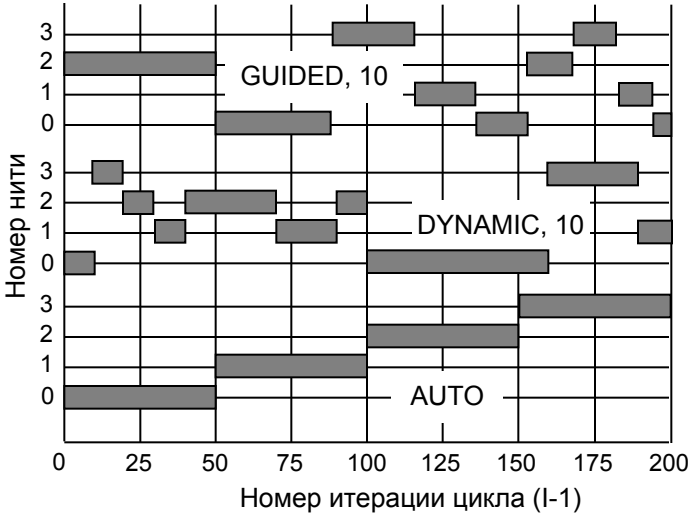
Директивы параллелизации циклов

```
 PROGRAM OPENMP_SCHEDULE
      INCLUDE 'omp_lib.h'
      INTEGER I, N, TID
      PARAMETER (N=9)
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(RUNTIME)
      DO I= 1, N
        PRINT *, 'Iteration', I, 'executed by thread ', TID
C       ...SOME WORK WITH ARRAY TO BE DONE...
      END DO
!$OMP END DO
!$OMP END PARALLEL
      END
```



Пример выполнения программы:

```
% export OMP_NUM_THREADS=4
% export OMP_SCHEDULE="STATIC,2"
% ./omp_schedule
Iteration 3 executed by thread 1
Iteration 4 executed by thread 1
Iteration 1 executed by thread 0
...
```



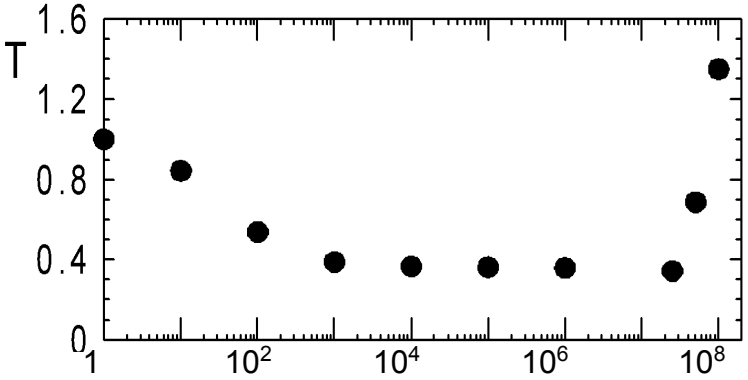
Показаны картины распределения итераций распараллеленного цикла при использовании разных значений атрибута SCHEDULE.

```
      PROGRAM OPENMP_PI_DO
      INTEGER N, I
      DOUBLE PRECISION X, STEP, PI, SUM /0.0/
      PARAMETER (N=100000000)
      STEP = 1.0/N
!$OMP PARALLEL PRIVATE(X)
!$OMP DO REDUCTION(+:SUM)
      DO I=1, N
        X = (I-0.5)*STEP
        SUM = SUM + 4.0/(1.0+X*X)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      PI = STEP * SUM
      END
```

Вариант программы вычисления определенного интеграла с использованием директивы параллелизации цикла и операции редукции для сложения локальных частичных сумм, рассчитанных группой нитей.

```
      PROGRAM OPENMP_DAXPY_DO
      INTEGER I, N
      PARAMETER (N = 100000000)
      DOUBLE PRECISION S /0.1/
      DOUBLE PRECISION, ALLOCATABLE :: Y(:), X(:)
      ALLOCATE (Y(N), X(N))
      X(1:N) = 1.0
      Y(1:N) = 2.0
!$OMP PARALLEL IF(N .GE. 100)
!$OMP DO SCHEDULE(RUNTIME)
      DO I=1, N
        Y(I) = Y(I) + S*X(I)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      DEALLOCATE (X, Y)
      END
```

Зависимость времени выполнения параллельной области программы от размера назначаемого нитям блока итераций (CHUNK_SIZE)



Вычислительные эксперименты выполнялись для 4-х нитей при значении $N=10^9$

Параллелизация циклов: зависимость по данным и последовательное выполнение итераций

Корректная параллелизация цикла возможна только при выполнении условия отсутствия внутри него **зависимости обрабатываемых данных**. В таком цикле любой промежуточный и окончательный результаты вычислений не должны зависеть от порядка выполнения итераций.

```
FACTORIAL(1)=1
!$OMP PARALLEL SHARED(FACTORIAL)
!$OMP DO
      DO I=2, N
      PRINT *, I
      FACTORIAL(I) = I * FACTORIAL(I-1)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      PRINT *, 'FACTORIAL=', FACTORIAL(N)
```

Результаты выполнения программы (для N=6):

```
% export OMP_NUM_THREADS=1
% ./omp_factorial
2 3 4 5 6
FACTORIAL=720
% export OMP_NUM_THREADS=2
% ./omp_factorial
5 6 2 3 4
FACTORIAL=0
```

Атрибут **ORDERED** (совместно с одноименной директивой синхронизации) позволяет организовать строго последовательное выполнение итераций цикла нитями, соответствующее порядку роста индекса.

```
 FACTORIAL(1)=1
!$OMP PARALLEL SHARED(FACTORIAL)
!$OMP DO ORDERED
      DO I=2, N
!$OMP ORDERED
      PRINT *, I
      FACTORIAL(I) = I * FACTORIAL(I-1)
!$OMP END ORDERED
      END DO
!$OMP END DO
!$OMP END PARALLEL
      PRINT *, 'FACTORIAL=', FACTORIAL(N)
```

Результаты выполнения программы (для N=6):

```
% export OMP_NUM_THREADS=2
% ./omp_factorial
2 3 4 5 6
FACTORIAL=720
```

Параллелизация циклов: отмена барьерной синхронизации

При завершении обработки структурного блока, ограниченного директивой распараллеливания циклов, по умолчанию производится неявная барьерная синхронизация нитей. Отменить ее можно путем указания с завершающей частью директивы распараллеливания цикла атрибута **NOWAIT**.

Данный атрибут целесообразно применять в том случае, если вычисленные в цикле переменные больше нигде в текущей параллельной области не задействуются или когда в коде программы друг за другом следуют два цикла, в которых производятся несвязанные между собой по данным вычисления.

```
...
!$OMP PARALLEL
!$OMP DO
      DO I=2, N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO I=1, N
        C(I) = SQRT(A(I))
      END DO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
...
```

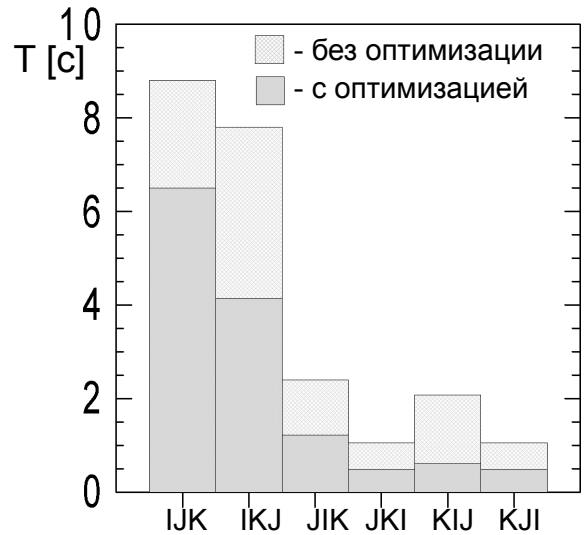
Параллелизация многоуровневых циклов

```
 PROGRAM OPENMP_3D_ARRAY
      INCLUDE 'omp_lib.h'
      INTEGER I, J, K, N
      PARAMETER (N=600)
      DOUBLE PRECISION T1, T2
      DOUBLE PRECISION, ALLOCATABLE :: A(:,:,:)
      ALLOCATE (A(N,N,N))
      ...ARRAY INITIALIZATION...
      T1 = OMP_GET_WTIME()
!$OMP PARALLEL
!$OMP DO
      DO K=1, N
       DO J=1, N
        DO I=1, N
            A(I,J,K) = A(I,J,K)/2.0
        END DO
       END DO
      END DO
!$OMP END DO
!$OMP END PARALLEL
      T2 = OMP_GET_WTIME()
      PRINT *, 'computational time: ', T2-T1, 's'
      END
```

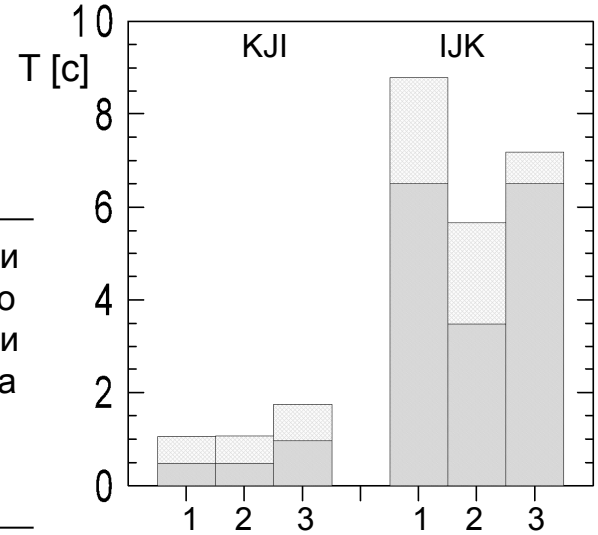
Время обработки многоуровневых (тесно вложенных) циклов при последовательном и параллельном выполнении программы существенно зависит от порядка вложенности и используемой опций оптимизации компилятора, а в параллельном режиме дополнительно – от места расположения директивы параллелизации цикла.

```
% gfortran -O2 -fopenmp -o pi_openmp pi_openmp.f
```

Время выполнения циклов для разных порядков вложенности



Время выполнения циклов для трех разных вариантов расположения директивы параллелизации



Параллелизация многоуровневых циклов: объединение наборов итераций

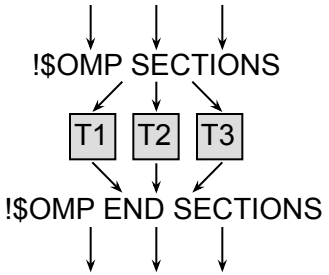
Атрибут `COLLAPSE` определяет особенности распараллеливания последовательных тесно вложенных циклов с прямоугольным индексным пространством, для которых не используются директивы вложенного параллелизма.

Параметр атрибута (целое положительное число) указывает на то, сколько петель цикла будут образовывать общее пространство итераций, совместно распределяемое реализацией между работающими нитями. В отсутствие этого атрибута действие директивы распараллеливания цикла распространяется только на непосредственно следующую за ней петлю цикла.

```
...
!$OMP PARALLEL PRIVATE(I, J, K)
!$OMP DO COLLAPSE(3)
      DO K=1, N
       DO J=1, N
        DO I=1, N
            A(I,J,K) = A(I,J,K)/2.0
        END DO
       END DO
      END DO
!$OMP END DO
!$OMP END PARALLEL
...
```

Неитеративное распределение работы между нитями

Директива SECTIONS применяется для объявления фрагментов кода внутри параллельной области программы, выполнение которых целиком (в виде независимых подзадач) будет распределено между разными нитями.




```
!$OMP SECTIONS [CLAUSE[[,] CLAUSE] ...]
[!$OMP SECTION]
  STRUCTURED-BLOCK
[!$OMP SECTION
  STRUCTURED-BLOCK]
...
!$OMP END SECTIONS [NOWAIT]
```

Возможные атрибуты:

```
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
LASTPRIVATE(LIST)
REDUCTION(OPERATOR:LIST)
```

Данная директива представляет собой неитеративную параллельную конструкцию, определяющую внутри параллельной области набор независимо и однократно исполняемых частей кода (параллельных секций).

Часть кода программы, соответствующая конкретной секции, исполняется целиком одной нитью из группы. Методика планирования обработки секций определяется конкретной реализацией.

Параллельное выполнение операции с трехмерной расчетной сеткой

```
 !$OMP PARALLEL
 !$OMP SECTIONS
 !$OMP SECTION
       CALL X_AXIS()
 !$OMP SECTION
       CALL Y_AXIS()
 !$OMP SECTION
       CALL Z_AXIS()
 !$OMP END SECTIONS
 !$OMP END PARALLEL
```

Параллельное чтение независимых данных из нескольких файлов

```
!$OMP PARALLEL SECTIONS SHARED(A, B) PRIVATE(I)
!$OMP SECTION
      READ(1,*) (A(I), I=1,N)
!$OMP SECTION
      READ(2,*) (B(I), I=1,N)
!$OMP END PARALLEL SECTIONS
```

Выполнение части кода одной нитью: директива SINGLE

Директива `SINGLE` применяется для выделения части кода внутри параллельной области, которую необходимо исполнить только один раз любой из нитей (первой, которая встретит эту директиву).

Все остальные нити из группы будут приостановлены до момента окончания выполнения ограниченного областью действия директивы структурного блока (если не указан отменяющий ожидание атрибут `NOWAIT`).

```
!$OMP SINGLE [CLAUSE[[,] CLAUSE] ...]
  STRUCTURED-BLOCK
!$OMP END SINGLE [END_CLAUSE[[,] END_CLAUSE] ...]
```

Возможные атрибуты:

```
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
COPYPRIVATE(LIST) - в Fortran - в качестве END_CLAUSE
NOWAIT - в Fortran - в качестве END_CLAUSE
```

Фрагмент кода параллельной программы с включением директивы SINGLE

```
!$OMP PARALLEL
!$OMP SINGLE
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Beginning WORK1() using ', NTHREADS, ' threads'
!$OMP END SINGLE
      CALL WORK1()
!$OMP SINGLE
      PRINT *, 'All threads finishing WORK1() and beginning WORK2()'
!$OMP END SINGLE
      CALL WORK2()
!$OMP END PARALLEL
```

Директива WORKSHARE

Директива WORKSHARE применяется только в языке Fortran осуществляет разделение соответствующего структурного блока на независимые единицы работы, каждая из которых будет целиком и однократно выполнена только одной нитью из группы.

Предполагается, что в случае наличия зависимости данных в выполняющихся разными нитями наборах операторов, в код программы системой поддержки реализации стандарта будут неявно добавлены все необходимые синхронизирующие конструкции.

```
!$OMP WORKSHARE
  STRUCTURED-BLOCK
!$OMP END WORKSHARE [NOWAIT]
```

Ограниченный областью директивы структурный блок может содержать только следующие конструкции:

- операции присваивания массивов и скалярных переменных
- операторы `FORALL`, `WHERE` (стандарт языка Fortran 90/95)
- директивы OpenMP - `PARALLEL`, `ATOMIC`, `CRITICAL`

<u>Пример использования директивы</u>

```
!$OMP PARALLEL
!$OMP WORKSHARE
      A(1:N) = A(1:N) / 2.0
      WHERE (B(1:N) /= 0.0)
        B(1:N) = SQRT(B(1:N))
      ELSEWHERE
        B(1:N) = 0.0
      END WHERE
      C(1:N) = A(1:N)+B(1:N)
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

<u>Вычисление произведения диагональных элементов матрицы</u>

```
!$OMP PARALLEL
!$OMP WORKSHARE
      DIAG(1:N) = 1.0
      FORALL (I=1:N, A(I,I) /= 0) DIAG(I)= A(I,I)
      DIAGPROD=PRODUCT(DIAG)
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

Составные директивы параллелизации и распределения работы между нитями

Составные (комбинированные) директивы OpenMP могут применяться в программе том случае, если внутри параллельной области содержится единственная директива распределения работы между нитями и соответствующий ей структурный блок.

```
!$OMP PARALLEL DO [CLAUSE[[,] CLAUSE] ...]]
  DO-LOOPS
[!$OMP END PARALLEL DO]


!$OMP PARALLEL SECTIONS [CLAUSE[[,] CLAUSE]...]
[!$OMP SECTION]
  STRUCTURED-BLOCK
[!$OMP SECTION
  STRUCTURED-BLOCK]
...
!$OMP END PARALLEL SECTIONS


!$OMP PARALLEL WORKSHARE [CLAUSE[[,] CLAUSE]...]
  STRUCTURED-BLOCK
!$OMP END PARALLEL WORKSHARE
```

Составные директивы по сравнению с раздельным написанием позволяют снизить накладные расходы на системные операции, выполняющиеся при последовательной трансляции соответствующих OpenMP-конструкций, а также сократить размер кода программы.

<u>Пример использования составной директивы</u>

```
!$OMP PARALLEL DO IF(N .GE. 100)
      DO I=1, N
        Y(I) = Y(I) + S*X(I)
      END DO
!$OMP END PARALLEL DO
```

Работа с явными задачами (1)

В актуальных спецификациях OpenMP работа нитей представляется в терминах модели ***задач*** (или заданий, ***tasks***). Параллельная область трактуется как совокупность выполняемых нитями *неявных задач* (*implicit tasks*), в контексте которых с помощью директив распределения работы и синхронизации нитей могут выполняться разные фрагменты кода и обрабатываться разные блоки данных.

OpenMP позволяет явным образом сформулировать и назначить нитям независимые задачи.
Явная задача (*explicit task*) - исполняемый нитью блок операторов, задействованные в них данные и набор внутренних переменных OpenMP. Динамически формируемый в параллельной области пул задач в соответствии с определенными правилами выполняется группой нитей.

Задачи можно рассматривать как "облегченную" (*lightweight*) реализацию нитей, ориентированную на выполнении множества относительно небольших наборов операций (более тонкая балансировка загрузки). Модель задач предоставила возможность параллелизации нерегулярных алгоритмов, в которых используются безграничные циклы (`DO WHILE`), рекурсивные функции (списки, графы и т.д.).

```
!$OMP TASK [CLAUSE[[,] CLAUSE] ...]
  STRUCTURED-BLOCK
!$OMP END TASK
```

<u>Возможные атрибуты:</u>

```
IF(SCALAR-EXPRESSION)
FINAL(SCALAR-EXPRESSION)
DEFAULT(TYPE)
MERGEABLE
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
SHARED(LIST)
UNTIED
DEPEND
```

Явная задача состоит из операторов структурированного блока. Для выполнения задачи система назначает одну из группы нитей, ориентируясь на степень ее готовности, незанятости выполнением других задач.

Выполнение созданной задачи может быть начато немедленно или отложено - такая задача будет помещена в воображаемое *хранилище задач*, ассоциированное с текущей параллельной областью. По мере освобождения от работы нити будут извлекать из этой очереди и выполнять всё новые задачи, пока хранилище полностью не опустеет.

Работа с явными задачами (2)

Связанная (*tied*) задача – соответствующей задаче программный код целиком, от начала до конца выполняется одной и той же нитью.

Несвязанная (*untied*) задача – код может быть выполнен по частям (в последовательном порядке) более чем одной нитью (атрибут `UNTIED`).

Нить может приостановить выполнение задачи, а через некоторое время снова возобновить его, либо переключиться на другую задачу. Если приостановленная задача является *связанной*, ее выполнение будет продолжено той же нитью. Выполнение приостановленной *несвязанной* задачи может продолжить любая нить из группы.

Атрибут `IF` позволяет задать условие, при неудовлетворении которого задача будет рассматриваться как "*срочная*" (***неотложная***, *undeferred*): встретившая директиву нить должна будет приостановить решение других своих задач и мгновенно переключиться на исполнение данной. Приостановленные задачи могут быть возобновлены только после полного завершения выполнения срочной задачи.

Присоединенная (*included*) задача – неотложная задача, запрос на создание которой происходит из кода выполняющейся явной задачи. Атрибут `FINAL` задает логическое условие, при удовлетворении которого задача помечается как ***заключительная*** (*финальная*, *final*): для такой задачи все ее задачи-потомки будут считаться присоединенными и выполняться последовательно.

Атрибут `MERGEABLE` – вместо задачи, которая должна быть создана как неотложная или присоединенная может быть сгенерирована задача специального вида - ***совмещенная*** (*merged*). Выполнение соответствующего структурного блока происходит как будто директива `TASK` вообще не была установлена (код такой задачи "погружается" в код родительской задачи).

Работа с явными задачами (3)

Директива `TASKYIELD` указывает, что текущая задача может быть приостановлена с целью переключения соответствующей нити на выполнение другой задачи.

Явно назначенные задачи по умолчанию выполняются в случайном порядке. Для синхронизации задач может быть использована директива `TASKWAIT`. Встретив эту директиву в собственном коде, задача приостанавливается до тех пор, пока все созданные ею к данному моменту дочерние задачи полностью не завершат выполнение предписанных им операций.

Диспетчеризацию работы явных задач разрешается производить в специальных *точках планирования* (*task scheduling point*), в которых код задачи может быть динамически разделен на несколько фрагментов. Точка планирования - место внутри кода задачи, в которой она может быть приостановлена для начала или возобновления выполнения другой задачи. В качестве точек планирования могут выступать директивы `TASK`, `TASKYIELD`, `TASKWAIT`, позиции явного и неявного назначения барьерной синхронизации, а также место завершения кода несвязанной задачи.

Работа с явными задачами: пример программы

Параллельная OpenMP-программа нахождения чисел Фибоначчи

```
      PROGRAM OPENMP_FIBONACCI
      INTEGER N, NUM, RES, FIB
      PARAMETER(N=40)
!$OMP PARALLEL
!$OMP SINGLE
      NUM = FIB(N)
!$OMP END SINGLE NOWAIT
!$OMP END PARALLEL
      PRINT *, NUM
      END

RECURSIVE INTEGER FUNCTION FIB(N) RESULT(RES)
      INTEGER N, X, Y, LIMIT
      PARAMETER(LIMIT=12)
      IF (N .LT. 2) THEN
        RES = N
      ELSEIF (N .LT. LIMIT) THEN
        RES = FIB(N-1)+FIB(N-2)
      ELSE
!$OMP TASK SHARED(X)
        X = FIB(N-1)
!$OMP END TASK
!$OMP TASK SHARED(Y)
        Y = FIB(N-2)
!$OMP END TASK
!$OMP TASKWAIT
        RES = X + Y
      END IF
      END FUNCTION
```

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}.$$

Числа Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

При разработке программы с помощью функций специального вида реализуется рекурсивный алгоритм.

В пределах рекурсивной функции `FIB()` на каждом шаге рекурсии порождается две новые явные задачи (или инициируются новые вызовы функций), до тех пор, пока аргумент `N >=2`. Перед определением результата вычислений с помощью директивы `TASKWAIT` выполняется синхронизация сгенерированных задач.

Для балансировки затрат на накладные расходы по организации задач и на вычисления применяется дополнительная проверка текущего номера шага рекурсии `N`: при `N < LIMIT` генерация новых задач в рекурсивном алгоритме прекращается и вычисления начинают производиться последовательно.

Работа с явными задачами: атрибут DEPEND

Атрибут `DEPEND` накладывает дополнительные ограничения на планирование задач и состоит из параметра `DEPENDENCE-TYPE` и одного или нескольких элементов списка.

```
!$OMP TASK DEPEND(DEPENDENCE-TYPE : LIST)
```

`IN DEPENDENCE-TYPE`

Создаваемая задача будет зависимой от всех ранее созданных «сестринских» (*sibling*) задач (такая задача является потомком задачи из того же структурированного блока), которая ссылается по крайней мере на одну из элементов списка в `OUT` или `INOUT` типа `DEPENDENCE-TYPELIST`.

`OUT DEPENDENCE-TYPE`

`INOUT DEPENDENCE-TYPE`

Создаваемая задача будет зависимой от всех ранее созданных «сестринских» (*sibling*) задач, которая ссылается по крайней мере на одну из элементов списка `IN`, `OUT` или `INOUT` типа `DEPENDENCE-TYPELIST`.

Работа с явными задачами: атрибут DEPEND

Пример использования явных задач и атрибута `DEPEND` в подпрограмме блочного перемножения плотных квадратных матриц.

```
      SUBROUTINE MATMUL_DEPEND (N, BS, A, B, C)
      INTEGER :: N, BS, BM, I, J, K, II, JJ, KK
      REAL, DIMENSION(N, N) :: A, B, C
      BM = BS -1
      DO I = 1, N, BS
        DO J = 1, N, BS
          DO K = 1, N, BS
!$OMP TASK DEPEND ( IN: A(I:I+BM, K:K+BM), B(K:K+BM, J:J+BM) ) &
!$OMP DEPEND ( INOUT: C(I:I+BM, J:J+BM) )
            DO II = I, I+BS
              DO JJ = J, J+BS
                DO KK = K, K+BS
                  C(JJ,II) = C(JJ,II) + A(KK,II) * B(JJ,KK)
                END DO
              END DO
            END DO
!$OMP END TASK
          END DO
        END DO
      END DO
      END SUBROUTINE
```

Директивы синхронизации в OpenMP

Принципиальнейшая задача при создании многопоточных приложений состоит в регулярно возникающей необходимости обеспечения синхронизированного, то есть согласованного неконкурентного доступа нитей к общей памяти, к хранящимся в ней общим переменным. Для этих целей API OpenMP предоставляет разработчикам представительный набор специальных конструкций - директив и библиотечных подпрограмм.

Наиболее общими подходами к обеспечению синхронизации считаются:

- *взаимное исключение* (*mutual exclusion*)
- *синхронизация по событию* (*event synchronization*)

Барьерная синхронизация: директива BARRIER

```
!$OMP BARRIER
```

Директива `BARRIER` явным образом задает в параллельной области точку *барьерной синхронизации* - место в программе, достигнув которого каждая нить из группы останавливается и дожидается всех остальных.

Параллельная версия программы "Hello, world!" с применением директивы BARRIER

```
 PROGRAM OPENMP_HELLO_BARRIER
      INCLUDE 'omp_lib.h'
      INTEGER N, TID
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello, world from thread ', TID
!$OMP BARRIER
      IF (TID .EQ. 0) THEN
        N = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads: ', N
      END IF
!$OMP END PARALLEL
      END
```

Директивы синхронизации в OpenMP

Выполнение части кода главной нитью: директива MASTER

```
!$OMP MASTER
  STRUCTURED-BLOCK
!$OMP END MASTER
```

Директива `MASTER` предназначена для выделения фрагмента кода программы внутри параллельной области, который будет целиком и последовательно исполнен главной нитью.
Остальные нити из группы пропустят эту часть кода и сразу перейдут к следующему за ним оператору (неявная барьерная синхронизация в конце области действия директивы не производится).

Параллельная версия программы "Hello, world!"
с применением директивы MASTER

```
 PROGRAM OPENMP_HELLO_MASTER
      INCLUDE 'omp_lib.h'
      INTEGER NTHREADS, TID
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello, world from thread ', TID
!$OMP MASTER
       NTHREADS = OMP_GET_NUM_THREADS()
       PRINT *, 'Number of threads: ', NTHREADS
!$OMP END MASTER
!$OMP END PARALLEL
      END
```

Упорядоченное выполнение части кода: директива ORDERED

```
!$OMP ORDERED
  STRUCTURED-BLOCK
!$OMP END ORDERED
```

Директива `ORDERED` назначает фрагмент кода внутри тела распараллеленного цикла, который должен обрабатываться в том порядке, который имел бы место при его последовательном выполнении (в порядке роста индексов). Директива параллелизации цикла должна при этом содержать опцию `ORDERED`.

Директивы синхронизации в OpenMP

Критические секции: директива CRITICAL

Директива `CRITICAL` реализует в OpenMP синхронизирующую операцию взаимного исключения. Данная директива определяет в параллельной области программы *критическую секцию* - часть кода, которая не может выполняться одновременно двумя и более нитями. В каждый момент времени в связанном с критической секцией структурном блоке может находиться только одна нить. Другие нити, достигнув начала критической секции, будут ожидать завершения выполнения этой нитью кода программы, соответствующего структурному блоку ("освобождения" критической секции), после чего одна из ожидающих нитей, выбранная случайным образом, "займет" критическую секцию.
Неявная барьерная синхронизация в конце области действия директивы не производится, а побочные входы/выходы из критической секции запрещены.

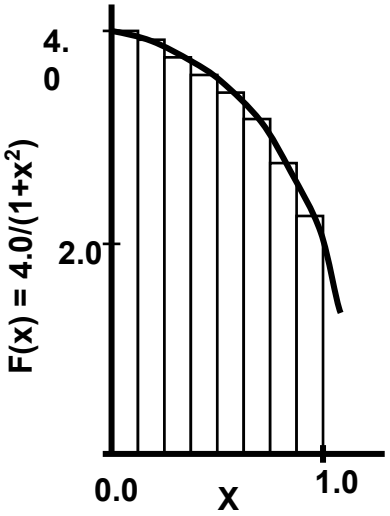
```
!$OMP CRITICAL [(NAME)]
  STRUCTURED-BLOCK
!$OMP END CRITICAL [(NAME)]
```

<u>Подсчет числа нитей в параллельной области программы с помощью директивы CRITICAL</u>

```
 PROGRAM OPENMP_NTHREADS_CRITICAL
      INCLUDE 'omp_lib.h'
      INTEGER N /0/, TID
!$OMP PARALLEL
!$OMP CRITICAL
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello, world from thread ', TID
      N=N+1
!$OMP END CRITICAL
!$OMP END PARALLEL
      PRINT *, 'Number of threads: ', N
      END
```

Параллелизация программы вычисления определенного интеграла (использование критической секции)

```
 PROGRAM OPENMP_PI_CRITICAL
      INTEGER I, N
      DOUBLE PRECISION X, STEP, PI /0.0/, THREAD_SUM /0.0/
      N = 100000000
      STEP = 1.0/N
!$OMP PARALLEL PRIVATE(X) FIRSTPRIVATE(THREAD_SUM)
!$OMP DO
      DO I=1, N
        X = (I-0.5)*STEP
        THREAD_SUM = THREAD_SUM + 4.0/(1.0+X*X)
      END DO
!$OMP END DO
!$OMP CRITICAL
      PI = PI + STEP * THREAD_SUM
!$OMP END CRITICAL
!$OMP END PARALLEL
      END
```



Директивы синхронизации в OpenMP

Атомарное обновление общих переменных: директива ATOMIC

Директива `ATOMIC` обеспечивает *атомарное* обновление заданной области оперативной памяти, т.е. гарантируется, что следующая непосредственно за директивой операция над общей переменной будет выполняться непрерывно, как единое целое, и на время выполнения операции к этой переменной будет заблокирован доступ на чтение и запись со стороны других нитей.

```
!$OMP ATOMIC
  STATEMENT

STATEMENT – выражение в одной из следующих форм:
X = X OPERATOR EXPR
X = EXPR OPERATOR X
X = INTRINSIC_PROCEDURE_NAME (X, EXPR_LIST)
X = INTRINSIC_PROCEDURE_NAME (EXPR_LIST, X)

EXPR – скалярное выражение, не содержащее переменной X;
EXPR_LIST – разделенный запятыми список скалярных выражений, не содержащих переменной X;
INTRINSIC_PROCEDURE_NAME – одна из встроенных процедур языка: MAX, MIN, IAND, IOR, IEOR
OPERATOR – одна из атомарных операций: +, *, -, /, .AND., .OR., .EQV., .NEQV.
```

<u>Подсчет числа нитей в параллельной области программы с помощью директивы ATOMIC</u>

```
 PROGRAM OPENMP_NTHREADS_ATOMIC
      INTEGER N /0/
!$OMP PARALLEL
!$OMP ATOMIC
      N=N+1
!$OMP END PARALLEL
      PRINT *, 'Number of threads: ', N
      END
```

Директивы синхронизации в OpenMP

Директива FLUSH

Модель работы с памятью, реализуемая в OpenMP, позволяет нитям иметь собственное представление, временный образ глобальной разделяемой памяти. Значения общих переменных в этом образе и в основной памяти могут отличаться. Для обеспечения согласованного состояния (одинакового вида) временной памяти нитей с общей памятью в многопоточных приложениях используется специальная синхронизирующая конструкция – операция сброса (*flush*). Выполнение этой операции может быть инициировано реализацией неявно, совместно с рядом других конструкций OpenMP. Разработчик программы может задействовать для этих целей OpenMP-директиву `FLUSH`, явно указав ее в коде параллельной области.

```
!$OMP FLUSH [(LIST)]
```

Директива явным образом определяет точку, в которой для вызвавшей ее нити должно быть обеспечено согласованное состояние оперативной памяти (всей или для набора переменных, указанных в качестве параметров). Исполнение директивы предусматривает, что переменные из регистров будут записаны в память, сбросятся буферы записи и т.д.

Данная синхронизирующая конструкция является весьма накла́дной (особенно при вызове без параметров) и ее неоправданное использование может негативно сказаться на показателях эффективности параллельной программы.

Библиотечные подпрограммы и переменные окружения OpenMP

Библиотечные подпрограммы и переменные окружения

Библиотечные подпрограммы OpenMP позволяют задавать и осуществлять контроль над параметрами среды исполнения программы применительно к особенностям обработки параллельных областей (количество порождаемых нитей, способ распределения между нитями итераций в распараллеленных циклах и др.).

Возможность вызова библиотечных подпрограмм OpenMP обеспечивается при явном подключении в коде программы имени соответствующей библиотеки (заголовочного файла):

<u>Язык Fortran</u>

```
INCLUDE 'omp_lib.h'
```

<u>Язык C/C++</u>

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

Переменные окружения OpenMP служат для создания переносимой среды запуска параллельных программ. Использование переменных окружения позволяет контролировать поведение программы, обходясь без ее перекомпиляции, в случаях, когда требуется изменить количество нитей, выполняющих параллельные области программы, задать тот или иной способ распределения итераций между нитями в распараллеленных циклах, разрешить или запретить вложенный параллелизм и т.д.

Переменные окружения в значительной степени дублируют функционал библиотечных подпрограмм времени исполнения OpenMP. Библиотечные подпрограммы OpenMP при одновременном использовании имеют приоритет над переменными окружения, а параметры директив, в свою очередь, имеют приоритет над библиотечными подпрограммами.

Установка переменных окружения в UNIX-подобных операционных системах:

```
% export OMP_NUM_THREADS=4
% setenv OMP_NUM_THREADS 4
% echo $OMP_NUM_THREADS
4
```

Подпрограммы для работы со временем

Функция OMP_GET_WTIME

Язык Fortran

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

Язык C/C++

```
double omp_get_wtime(void);
```

Данная функция возвращает время (в секундах), прошедшее с некоторого момента в прошлом до точки ее вызова (“точка отсчета” - полночь 1 января 1970 г.). Для измерения времени работы некоторого фрагмента программы его следует окружить вызовами этой функции, при этом разность возвращаемых ею значений даст искомое время. Синхронизация работы таймеров нитей стандартом не гарантируется, поэтому результаты замера времени разными нитями могут отличаться.

```
DOUBLE PRECISION START_TIME, END_TIME
...
START_TIME = OMP_GET_WTIME()
...WORK TO BE TIMED...
END_TIME = OMP_GET_WTIME()
PRINT *, 'Working time: ', END_TIME-START_TIME, 'seconds’
```

Функция OMP_GET_WTICK

Язык Fortran

```
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
```

Язык C/C++

```
double omp_get_wtick(void);
```

Данная функция возвращает разрешение системного таймера, используемое процедурой `OMP_GET_WTIME()` и характеризующее его точность (количество секунд между последовательными импульсами таймера).

Подпрограммы для контроля за выполнением параллельной программы (1)

Установка количества нитей в параллельной области

Язык Fortran

```
SUBROUTINE OMP_SET_NUM_THREADS(NUM_THREADS)
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

Язык C/C++

```
void omp_set_num_threads(int num_threads);
int omp_get_num_threads(void);
```

Библиотечная процедура `OMP_SET_NUM_THREADS()` позволяет назначить максимальное число нитей, создаваемых при входе в следующую за ней параллельную область программы (если это число разрешено менять динамически).

Функция `OMP_GET_NUM_THREADS()` возвращает фактическое число нитей в группе, выполняющих текущую параллельную область программы.

Получение идентификатора нити

Язык Fortran

```
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

Язык C/C++

```
int omp_get_thread_num(void);
```

Данная функция возвращает порядковый номер (идентификатор) в группе вызвавшей ее нити.

Подпрограммы для контроля за выполнением параллельной программы (2)

Функция OMP_GET_NUM_PROCS

Язык Fortran

```
INTEGER FUNCTION OMP_GET_NUM_PROCS()
```

Язык C/C++

```
int omp_get_num_procs(void);
```

Функция возвращает количество физических вычислительных устройств (процессоров или ядер), ресурсы которых доступных программе в момент выполнения.

Функция OMP_GET_THREAD_LIMIT

Язык Fortran

```
INTEGER FUNCTION OMP_GET_THREAD_LIMIT()
```

Язык C/C++

```
int omp_get_thread_limit(void);
```

Функция возвращает определяемое реализацией стандарта максимально возможное число нитей, доступных для выполнения программы.

Подпрограммы для контроля за выполнением параллельной программы (3)

Функция OMP_GET_MAX_THREADS

Язык Fortran

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

Язык C/C++

```
int omp_get_max_threads(void);
```

Функция возвращает максимальное число нитей, которое может быть задействовано для выполнения последующей параллельной области программы.

Функция OMP_IN_PARALLEL

Язык Fortran

```
LOGICAL FUNCTION OMP_IN_PARALLEL()
```

Язык C/C++

```
int omp_in_parallel(void);
```

Логическая функция, которая возвращает значение "истина" (`.TRUE.` - в Fortran, `1` - в C/C++) в том случае, если она вызвана из параллельной области программы, иначе будет возвращено значение "ложь" (`.FALSE.`). Эту функцию полезно применять в параллельных программах со сложной модульной структурой, а также при наличии оторванных OpenMP-конструкций.

Подпрограммы для контроля за выполнением параллельной программы (4)

Динамический и статический режимы выполнения программы

Язык Fortran

```
SUBROUTINE OMP_SET_DYNAMIC (DYNAMIC_THREADS)
LOGICAL FUNCTION OMP_GET_DYNAMIC()
```

Язык C/C++

```
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic(void);
```

Данные подпрограммы устанавливают (запрашивают) состояние динамического выполнения последующей параллельной области программы, характеризующееся возможностью изменения системой количества создаваемых нитей.

Способ распределения итераций цикла

Язык Fortran

```
SUBROUTINE OMP_SET_SCHEDULE(KIND, MODIFIER)
SUBROUTINE OMP_GET_SCHEDULE(KIND, MODIFIER)
```

Язык C/C++

```
void omp_set_schedule(omp_sched_t kind, int modifier);
void omp_get_schedule(omp_sched_t * kind, int * modifier);
```

Данные подпрограммы устанавливают (запрашивают) текущий способ распределения итераций распараллеленного средствами OpenMP цикла между нитями. Возможные способы распределения итераций подробно рассматривались ранее.

Подпрограммы для контроля за выполнением параллельной программы (5)

Контроль режима вложенного параллелизма

Язык Fortran

```
SUBROUTINE OMP_SET_NESTED (NESTED)
LOGICAL FUNCTION OMP_GET_NESTED()
```

Язык C/C++

```
void omp_set_nested(int nested);
int omp_get_nested(void);
```

Данные подпрограммы устанавливают (запрашивают) состояние режима поддержки многоуровневого (вложенного) параллелизма, характеризующегося возможностью порождения дополнительных нитей для выполнения вложенных параллельных областей.

Подпрограммы OMP_SET_MAX_ACTIVE_LEVELS / OMP_GET_MAX_ACTIVE_LEVELS

Язык Fortran

```
SUBROUTINE OMP_SET_MAX_ACTIVE_LEVELS (MAX_LEVELS)
INTEGER FUNCTION OMP_GET_MAX_ACTIVE_LEVELS()
```

Язык C/C++

```
void omp_set_max_active_levels (int max_levels);
int omp_get_max_active_levels(void);
```

Данные подпрограммы устанавливают (запрашивают) максимально допустимое количество вложенных параллельных областей в программе. Максимально возможное значение параметра определяется реализацией стандарта и не может быть превышено при установке.

Подпрограммы для контроля за выполнением параллельной программы (6)

Функция OMP_GET_LEVEL

Язык Fortran

```
INTEGER FUNCTION OMP_GET_LEVEL()
```

Язык C/C++

```
int omp_get_level(void);
```

Функция возвращает для вызвавшей ее нити (выполняемой задачи) уровень вложенного параллелизма, соответствующий текущей параллельной области.

Функция OMP_GET_ANCESTOR_THREAD_NUM

Язык Fortran

```
INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(LEVEL)
```

Язык C/C++

```
int omp_get_ancestor_thread_num(int level);
```

Функция возвращает для вызвавшей ее нити и заданного параметром уровня вложенного параллелизма номер нити-родителя.

Подпрограммы для контроля за выполнением параллельной программы (7)

Функция OMP_GET_TEAM_SIZE

Язык Fortran

```
INTEGER FUNCTION OMP_GET_TEAM_SIZE(LEVEL)
INTEGER LEVEL
```

Язык C/C++

```
int omp_get_team_size(int level);
```

Функция возвращает для заданного параметром уровня вложенного параллелизма количество нитей, порожденных нитью-родителем по отношению к нити, совершившей вызов данной функции.

Функция OMP_GET_ACTIVE_LEVEL

Язык Fortran

```
INTEGER FUNCTION OMP_GET_ACTIVE_LEVEL()
```

Язык C/C++

```
int omp_get_active_level(void);
```

Функция возвращает количество вложенных "активных" параллельных областей (выполняемых двумя и более нитями) в момент выполнения вызова

Процедуры синхронизации на основе переменных-замков

Представляемый далее набор библиотечных процедур OpenMP реализует операцию ***взаимного исключения нитей*** при помощи семафоров специального типа - ***переменных-замков (locks)***.

Процедуры синхронизации на базе замков обеспечивают большую гибкость и универсальность по сравнению директивами `CRITICAL` и `ATOMIC`. В качестве замков используются общие переменные типа `INTEGER`.

Инициализация/уничтожение замка

Язык Fortran

```
OMP_INIT_LOCK(VAR)
INTEGER (KIND=OMP_LOCK_KIND) VAR
OMP_DESTROY_LOCK(VAR)
```

Язык C/C++

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Захват/освобождение замка

Язык Fortran

```
OMP_SET_LOCK(VAR)
OMP_UNSET_LOCK(VAR)
```

Язык C/C++

```
omp_set_lock(omp_lock_t *lock);
omp_unset_lock(omp_lock_t *lock);
```

Процедуры синхронизации на основе переменных-замков

Проверка возможности захвата замка

Язык Fortran

```
OMP_TEST_LOCK(VAR)
```

Язык C/C++

```
omp_test_lock(omp_lock_t *lock)
```

Пример использования переменных замков:

```
PROGRAM OPENMP_PI_LOCK
      INCLUDE 'omp_lib.h'
      INTEGER I, N, TID
      INTEGER(OMP_LOCK_KIND) LOCK
      DOUBLE PRECISION X, STEP, PI /0.0/, THREAD_SUM /0.0/
      N= 100000000
      STEP = 1.0/N
      CALL OMP_INIT_LOCK(LOCK)
!$OMP PARALLEL PRIVATE(X) FIRSTPRIVATE(THREAD_SUM)
!$OMP DO
      DO I=1, N
        X = (I-0.5)*STEP
        THREAD_SUM = THREAD_SUM + 4.0/(1.0+X*X)
      END DO
!$OMP END DO
      CALL OMP_SET_LOCK(LOCK)
      PI = PI + STEP * THREAD_SUM
      CALL OMP_UNSET_LOCK(LOCK)
!$OMP END PARALLEL
      CALL OMP_DESTROY_LOCK(LOCK)
      END
```

Переменные окружения OpenMP

Имя переменной	Формат значения	Назначение переменной
OMP_NUM_THREADS	N (целое число)	число нитей, выполняющих параллельные области программы
OMP_SCHEDULE	TYPE[,CHUNK]	задание способа распределения между нитями итераций в распараллеленных циклах
OMP_DYNAMIC	TRUE | FALSE	разрешение/запрещение динамического изменения числа нитей, выполняющих параллельные области программы
OMP_NESTED	TRUE | FALSE	разрешение/запрещение вложенного параллелизма
OMP_MAX_ACTIVE_LEVELS	N (целое число)	задание максимально возможного числа активных вложенных параллельных областей
OMP_THREAD_LIMIT	N (целое число)	максимальное число нитей, используемое для выполнения программы
OMP_STACKSIZE	SIZE | SIZEB(K|M|G)	задание размера стека для порожденных нитей (без указания единицы измерения - в КБайт)
OMP_WAIT_POLICY	ACTIVE | PASSIVE	задание стратегии ("политики") поведения ожидающих исполнения нитей

Переменная окружения `OMP_STACKSIZE` позволяет задать размер локального для каждой из нитей стека, создаваемого и поддерживаемого средствами OpenMP (размер стека "по умолчанию" зависит от реализации).

Переменная окружения `OMP_WAIT_POLICY` служит для "тонкой" настройки выполнения программы с точки зрения особенностей поведения нитей. При установке для нее значения `ACTIVE` ожидающие исполнения нити должны быть преимущественно "активными“ (под этим понимается потребление определенного объема ресурсов процессора даже во время ожидания). В противном случае такие нити должны минимизировать потребление ресурсов (например, путем перевода в "спящее" состояние).

Некоторые вопросы повышения эффективности параллельных программ

Негативное влияние на показатели производительности могут оказывать следующие факторы:

- несбалансированность вычислительной загрузки
- неэффективная параллелизация
- необязательные синхронизирующие операции

Эффективная *балансировка загрузки* (load balancing) подразумевает обеспечение максимально равномерной вычислительной нагрузки процессоров (ядер) при выполнении распараллеленной программы. Возможны два подхода по управлению балансировкой загрузки: статический и динамический. В первом случае за обеспечение балансировки отвечает разработчик программы, заранее назначая нитям соответствующие, близкие по объемам наборы вычислительных операций. Динамическая балансировка загрузки предполагает наличие возможности перераспределения работы между нитями непосредственно в ходе выполнения программы. Например, завершившая свою работу нить может забрать себе часть операций у наиболее загруженной в данный момент нити (такая концепция реализована в модели задач OpenMP).

Примером *неэффективной параллелизации* является попытка параллельного выполнения коротких циклов, при которой издержки на работу с нитями могут «украсть» весь выигрыш от распараллеливания.

Наиболее часто встречающиеся ошибки в OpenMP-программах связаны с некорректной работой с общими переменными:

- конкуренция потоков при доступе к данным (условия состязательности, "гонка за данными", *race conditions*)
- взаимоблокировка нитей ("мертвая блокировка", *deadlock*).

Большой потенциал увеличения производительности программ связан с оптимизацией использования кэш-памяти. Эффективное и согласованное использование быстрой кэш-памяти и регистров процессоров в многопроцессорных вычислительных системах сопряжено с решением проблемы *когерентности*, под которой понимается коррекция общих данных, оказавшихся в момент их обработки размноженными и распределенными между кэшами разных процессоров.

Дополнительные примеры кодов программ (1)

Вычисление скалярного произведения векторов

$$r = x^T y = \sum_{i=1}^{n} x_i y_i$$

```
      PROGRAM OPENMP_DDOT_DO
      INTEGER I, N
      PARAMETER (N = 10000000)
      DOUBLE PRECISION R
      DOUBLE PRECISION, ALLOCATABLE ::
Y(:), X(:)
      ALLOCATE (Y(N), X(N))
      X(1:N) = 1.0
      Y(1:N) = 2.0
!$OMP PARALLEL DO REDUCTION(+:R)
      DO I=1, N
        R = R + X(I)*Y(I)
      END DO
!$OMP END PARALLEL DO
      DEALLOCATE (X, Y)
      END
```

Умножение матрицы на вектор

$$y_i = \sum_{j=1}^{n} A_{ij} \cdot x_j \ , \ i = 1,...,n$$

```
      PROGRAM OPENMP_DGEMV
      INTEGER I, J, N
      PARAMETER (N=10000)
      DOUBLE PRECISION, ALLOCATABLE :: Y(:),
A(:,:), X(:)
      ALLOCATE (Y(N), A(N,N), X(N))
      X(1:N) = 1.0
      DO I = 1, N
        A(I,1:N) = I
      END DO
!$OMP PARALLEL DO PRIVATE(I, J)
      DO I = 1, N
        Y(I) = 0.0
        DO J = 1, N
          Y(I) = Y(I) + A(I,J)*X(J)
        END DO
      END DO
!$OMP END PARALLEL DO
      DEALLOCATE(Y, A, X)
      END
```

Дополнительные примеры кодов программ (2)

Умножение квадратных матриц

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj} \ , \ \ i, j = 1,\ldots,n$$

```
      PROGRAM OPENMP_DGEMM
      INTEGER I, J, K, N, CHUNK
      PARAMETER (N=1000)
      DOUBLE PRECISION, ALLOCATABLE :: A(:,:), B (:,:), C(:,:)
      ALLOCATE (A(N,N), B(N,N), C(N,N))
      CHUNK=1
!$OMP PARALLEL PRIVATE(I, J, K)
!$OMP DO SCHEDULE(STATIC, CHUNK)
      DO I=1, N
        DO J=1, N
          A(I,J) = (I+J)/(I*J)
          B(I,J) = (I+J)/(I*J)
          C(I,J) = 0
        END DO
      END DO
!$OMP DO SCHEDULE(STATIC, CHUNK)
      DO I = 1, N
        DO J = 1, N
          DO K = 1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
          END DO
        END DO
      END DO
!$OMP END PARALLEL
      END
```

Дополнительные примеры кодов программ (3)

Распараллеливание операции LU-разложения матрицы

```
      PROGRAM OPENMP_DGETRF
      INTEGER I, J, K, N
      PARAMETER (N=1000)
      DOUBLE PRECISION, ALLOCATABLE :: A(:,:)
      ALLOCATE (A(N,N))
      CALL MATINIT(A) ! Вызов процедуры инициализации матриц
!$OMP PARALLEL IF (N .GE. 100) PRIVATE(I, J, K)
      DO K = 1, N
!$OMP SINGLE
        DO I = K+1, N
          A(I,K) = A(I,K)/A(K,K)
        END DO
!$OMP END SINGLE
!$OMP DO
        DO J = K+1, N
          DO I = K+1, N
            A(I,J) = A(I,J) - A(I,K)*A(K,J)
          END DO
        END DO
!$OMP END DO
      END DO
!$OMP END PARALLEL
      DEALLOCATE (A)
      END
```

Дополнительные примеры кодов программ (4)

Численное решение двумерного уравнения Пуассона по методу Якоби

$$\frac{\partial^2 w}{\partial x^2}+\frac{\partial^2 w}{\partial y^2}=-1 \qquad x, y \in [-1\,;1]$$

Применим упрощенную постановку задачи Дирихле: вычисления будем выполнять для области в форме квадрата `[-1;1]` с нулевыми граничными условиями. Используем ортогональную равномерную расчетную сетку, включающую `N×N` узлов.

$h = 2/(N-1)$ - шаг сетки

Формулировка линейной задачи в такой постановке дает, в частности, решение задачи об установившемся ламинарном движении вязкой несжимаемой жидкости сквозь трубу квадратного сечения. Искомая функция `w` представляет собой распределение продольной скорости по сечению трубы. Данная задача имеет аналитическое решение в виде разложения в бесконечные ряды, которое можно использовать для верификации разрабатываемой программы (в безразмерной форме):

$$w=\frac{16}{\pi^3}\cdot\sum_{p=0}^{\infty}\frac{(-1)^p}{(2n+1)^3}\left[1-\frac{ch\left(\frac{2p+1}{2}\cdot\pi y\right)}{ch\left(\frac{2p+1}{2}\cdot\pi\right)}\right]\cdot\cos\left(\frac{2p+1}{2}\cdot\pi x\right)$$

Пространственную дискретизацию уравнения Пуассона проведем по методу конечных разностей со вторым порядком точности

$$\frac{w_{i+1,j}-2w_{i,j}+w_{i-1,j}}{h^2}+\frac{w_{i,j+1}-2w_{i,j}+w_{i,j-1}}{h^2}=-1$$

Организуем и применим метод простой итерации (метод Якоби) для непосредственного нахождения значений функции во всех внутренних узлах сетки в соответствии с формулой (*l* - счетчик итераций):

$$w_{i,j}^{l+1}=0.25\cdot\left(w_{i+1,j}^{l}+w_{i-1,j}^{l}+w_{i,j+1}^{l}+w_{i,j-1}^{l}+h^2\right)$$

Дополнительные примеры кодов программ (4)

Численное решение двумерного уравнения Пуассона по методу Якоби: код программы

```
      PROGRAM OPENMP_POISSON
      PARAMETER (N=101)
      DOUBLE PRECISION, ALLOCATABLE :: W(:,:),WOLD(:,:)
      ALLOCATE (W(N,N), WOLD(N,N))
C Задание начальных и граничных условий
      CALL INIT(N, W)
C Решение уравнения итерационным методом Якоби с параллелизацией
      CALL JACOBI(N, W, WOLD)
      DEALLOCATE (W, WOLD)
      END

      SUBROUTINE INIT(N, W)
      INTEGER N
      DOUBLE PRECISION W(N,*)
      W(2:N-1,2:N-1) = 0.5
      W(1,1:N) = 0.0
      W(N,1:N) = 0.0
      W(1:N,1) = 0.0
      W(1:N,N) = 0.0
      END SUBROUTINE INIT
```

Дополнительные примеры кодов программ (4)

Численное решение двумерного уравнения Пуассона по методу Якоби: код программы

```
      SUBROUTINE JACOBI(N, W, WOLD)
      INTEGER N, L, MAXIT
      DOUBLE PRECISION W(N,*), WOLD(N,*)
      DOUBLE PRECISION ERROR, EPS, H
      MAXIT=5000
      EPS = 1E-6
      ERROR = 1E6
      L = 1
      H = 2.0/DBLE(N-1)
      DO WHILE (L .LE. MAXIT .AND. ERROR .GT. EPS)
        ERROR = 0.0
!$OMP PARALLEL PRIVATE(I,J)
!$OMP WORKSHARE
      WOLD(1:N,1:N) = W(1:N,1:N)
!$OMP END WORKSHARE
!$OMP DO REDUCTION(MAX:ERROR)
        DO J = 2, N-1
          DO I = 2, N-1
            W(I,J) = 0.25 * (WOLD(I-1,J) + WOLD(I+1,J)
     &            + WOLD(I,J-1) + WOLD(I,J+1) + H*H)
            ERROR = MAX(ERROR, ABS(W(I,J) - WOLD(I,J)))
          END DO
        ENDDO
!$OMP END DO
!$OMP END PARALLEL
        L = L + 1
      END DO
      RETURN
      END SUBROUTINE JACOBI
```

Приведенный код является одной из возможных реализаций многопоточного алгоритма Якоби. Для уменьшения накладных расходов можно внутри цикла `DO WHILE`, вынесенного в основную программу, организовать вызов процедуры-решателя `JACOBI()`, которая выполнит некоторое количество итерационных шагов (например, 100) без порождения новых нитей и проверки сходимости после каждого шага (распараллеливаться по-прежнему будет цикл вычисления новых значений функции). Проверку условия сходимости предлагается в этом случае проводить после выполнения заданного количества шагов, переходя затем, при необходимости, к новому блоку итераций.